Python 3

Apprendre à programmer en Python avec Pyzo et Jupyter Notebook

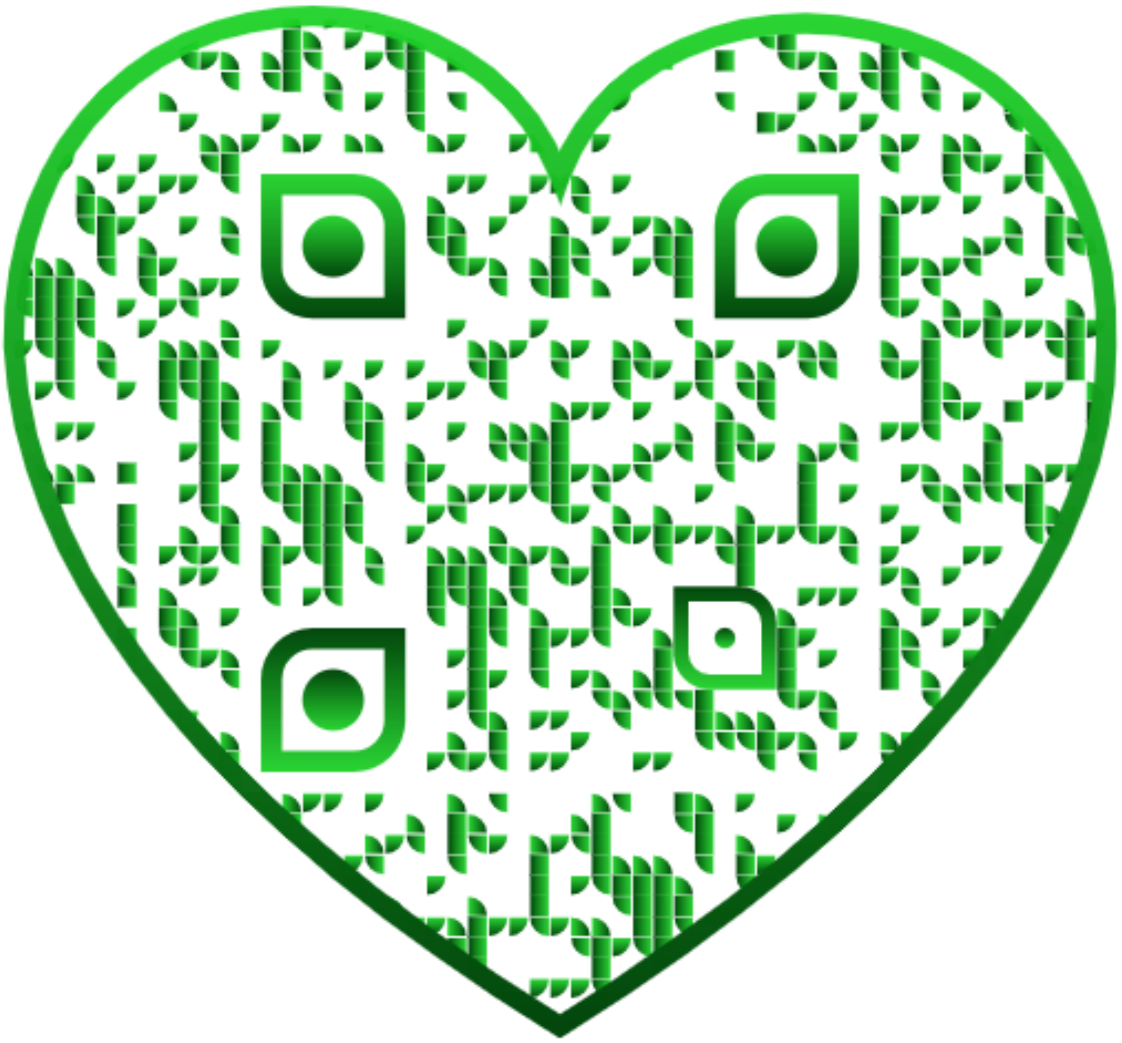
Bob Cordeau
Ancien ingénieur d'études à l'Onera
Ancien enseignant à l'Université Paris-Sud

Laurent Pointal
Informaticien au LIMSI/CNRS
Chargé de cours à l'Université Paris-Sud – IUT d'Orsay

Préface de **Gérard Swinnen**

Copyright © 2017 Dunod.

DUNOD



Toutes les marques citées dans cet ouvrage
sont des marques déposées par leurs propriétaires respectifs.

Illustrations intérieures :
© Hélène Cordeau

Illustration de couverture :
© Rachid Maraï

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, 2017

11, rue Paul Bert, 92240 Malakoff
www.dunod.com

ISBN 978-2-10-076870-7

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Copyright © 2017 Dunod.

Préface

Professeur de sciences désormais retraité de l'enseignement secondaire belge, je fus de ces aventuriers qui se lancèrent à la découverte des premiers micro-ordinateurs *grand public* à la fin des années 1970. Il fallait être un peu fou, à cette époque, pour investir des sommes plutôt rondelettes dans ces machines bricolées, au comportement assez capricieux, dont on fantasmait de tirer tôt ou tard des applications extraordinaires, mais souvent sans trop savoir au juste lesquelles, et encore moins comment on pourrait y arriver.

Il n'était évidemment pas question d'Internet en ce temps-là. Trouver de la documentation était une tâche ardue. Les rares documents que l'on parvenait à trouver (*via* les clubs de radio-amateurs, principalement) traitaient davantage d'électronique que de programmation. Et c'était bien nécessaire, il valait mieux être capable de manier le fer à souder ou de trouver l'un ou l'autre copain technicien dans un laboratoire disposant d'un programmateur d'EPROM [1].

C'est dans ce contexte que je découvris en autodidacte (inutile de dire qu'aucune formation n'était encore organisée à l'époque) mes premiers langages de programmation. Sur le TRS-80 de mes débuts, on disposait seulement d'un Basic sommaire et d'un Assembleur. Il fallait s'accrocher. La mise au point d'un tout petit programme pouvait prendre des heures, et même sa sauvegarde (sur cassette à bande magnétique !) pouvait se révéler problématique. Pas question en tout cas d'imaginer une seule seconde enseigner ce genre de choses à mes jeunes élèves.

Mon activité de programmation en ces années-là se focalisa alors sur le développement de simulations expérimentales. Sur le modèle anglo-saxon des années 1960, je souhaitais centrer mon enseignement scientifique sur la découverte et l'investigation personnelle des élèves, et j'organisais donc un maximum de séances de travaux pratiques. La simulation me permettait d'étendre cette méthodologie à des expérimentations cruciales pour la compréhension de principes fondamentaux (en physique ou en biologie, par exemple), mais irréalisables dans le cadre scolaire ordinaire pour des raisons diverses. Avec un programme de simulation d'expérience bien conçu, l'élève peut se trouver plongé dans une situation de travail très proche de celle d'un laboratoire. Je trouvais particulièrement intéressante, sur le plan pédagogique, l'idée qu'en procédant de la sorte j'instaurais pour l'étudiant un véritable droit à l'erreur : en simulation, il peut en effet décider lui-même sa stratégie expérimentale, procéder par tâtonnements, se tromper, recommencer éventuellement un grand nombre de fois ses tentatives, sans qu'il en résulte un coût excessif en temps ou en ressources matérielles.

Je progressais ainsi dans ma connaissance de la programmation, sans aucune intention de l'enseigner un jour, en m'adaptant au fil des années aux évolutions du matériel et des langages, jusqu'à ce jour de 1998 où l'on me demanda de participer à l'élaboration de cursus pour une nouvelle filière d'enseignement secondaire qui serait centrée sur l'apprentissage de l'informatique.

1. La mémoire EPROM (*Erasable Programmable Read-Only Memory*) est un type de mémoire morte reprogrammable. Pour effectuer cette (re)programmation, il faut en général retirer l'EPROM de son support et la placer dans un appareil dédié à cet effet.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Mon expérience et mes contacts m'avaient entre-temps fait prendre conscience de la problématique de la liberté logicielle. J'étais opportunément en train de découvrir l'une des premières distributions *crédibles* de Linux (l'une des premières Red Hat), et je me suis immédiatement persuadé que si l'on voulait effectivement inculquer une saine compréhension de ce que sont l'informatique et ses enjeux à des étudiants aussi jeunes, on se devait de le faire sur la base de logiciels libres.

L'un des cours à mettre en place devait être une initiation à la programmation. J'avais une certaine expérience en la matière, et c'était pour cela qu'on sollicitait mon avis, mais tous les outils que j'avais utilisés personnellement jusque-là étaient des langages propriétaires (Basic, Delphi, Clarion…), et je ne voulais être le démarcheur d'aucun d'entre eux. C'est donc dans cet esprit que je me suis mis à la recherche de ce que je craignais être la quadrature du cercle : un langage de programmation qui soit à la fois libre, multi-plateformes, polyvalent, assez facile à apprendre, avec lequel il soit possible d'aborder un maximum de concepts, tant sur les paradigmes de programmation que sur les structures de données, qui soit surtout de haut niveau et très lisible (je m'imaginais à l'avance le casse-tête que constituerait pour les professeurs le travail de correction d'un programme mal écrit par un élève à l'esprit tordu, dans un langage proche de la machine et à la syntaxe alambiquée…).

Le miracle a eu lieu : j'ai découvert Python. Ses qualités sont décrites dans les pages qui suivent.

Restait le problème de l'enseigner à des jeunes de 16-18 ans. En l'occurrence je souhaitais aussi valider autant que possible la stratégie pédagogique d'apprentissage par investigation libre que j'avais développée pour mes cours de sciences, et aucun cours de programmation satisfaisant à mes critères n'existait à l'époque, du moins en français. L'essentiel de la documentation de Python lui-même n'existait d'ailleurs qu'en anglais (on en était à la version 1.5).

Je me suis donc lancé le défi – encore une fois un peu fou – de rédiger mon propre manuel de cours. La suite est connue : bien conscient de mes limitations d'autodidacte, j'ai tout de suite mis mes notes à la disposition de tout le monde sur l'Internet, et j'ai ainsi pu récolter de nombreux avis et conseils, grâce auxquels le texte s'est amélioré au fil du temps et a fini par paraître aussi en version imprimée, distribuée en librairies.

C'est au cours de cette saga que j'ai eu la chance de faire la connaissance de Bob Cordeau, qui m'a gentiment rendu le service de relire mes 430 pages pour y débusquer coquilles et étourderies. Au cours de cet important travail, il a donc eu tout le loisir de constater tous les défauts de mon texte : imprécisions diverses, structuration fantaisiste, concepts omis ou traités de manière triviale…

Il ne m'en a rien dit pour ne pas me faire de la peine, mais il s'est courageusement mis à l'ouvrage pour rédiger son propre texte, que vous aurez le plaisir de découvrir dans les pages qui suivent. Là où je m'étais contenté d'une ébauche brouillonne, Bob et Laurent ont réalisé un vrai travail de « pro » : un des meilleurs textes de référence sur ce merveilleux outil qu'est Python.

Bonne lecture, donc.

Gérard Swinnen [1]

1. Auteur d'*Apprendre à programmer avec Python* 3, paru aux éditions Eyrolles, et disponible également en téléchargement libre sur http://inforef.be/swi/python.htm

Copyright © 2017 Dunod.

Table des matières

Préface v

Avant-propos xiii

1 Programmer en Python 1
- 1.1 Mais pourquoi donc apprendre à programmer ? 1
  - 1.1.1 Un exemple pratique 2
  - 1.1.2 Et après ? 6
- 1.2 Mais pourquoi donc apprendre Python ? 6
  - 1.2.1 Principales caractéristiques du langage Python 7
  - 1.2.2 Implémentations de Python 8
- 1.3 Comment passer du problème au programme 8
  - 1.3.1 Réutiliser 8
  - 1.3.2 Réfléchir à un algorithme 9
  - 1.3.3 Résoudre « à la main » 9
  - 1.3.4 Formaliser 9
  - 1.3.5 Factoriser 10
  - 1.3.6 Passer de l'idée au programme 10
- 1.4 Environnements matériel et logiciel 11
  - 1.4.1 L'ordinateur 11
  - 1.4.2 Deux sortes de programmes 11
- 1.5 Différents niveaux de langages 11
- 1.6 Les techniques de production des programmes 12
  - 1.6.1 Technique de production de Python 12
  - 1.6.2 Construction des programmes 13
- 1.7 Thèmes de réflexion 13

2 La calculatrice Python 15
- 2.1 Les deux modes d'exécution d'un code Python 15
- 2.2 Les commentaires 16
- 2.3 Identificateurs et mots clés 16
  - 2.3.1 Identificateurs 16
  - 2.3.2 Style de nommage 16
  - 2.3.3 Les mots réservés de Python 3 17
- 2.4 Notion d'expression 17
- 2.5 Les types de données entiers 17

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2.5.1 Le type `int` . . . 18
2.5.2 Le type `bool` . . . 19
2.6 Les types de données flottants . . . 20
2.6.1 Le type `float` . . . 20
2.6.2 Le type `complex` . . . 21
2.7 Variables et affectation . . . 21
2.7.1 Les variables . . . 21
2.7.2 L'affectation (ou assignation) . . . 22
2.7.3 Attention : affecter n'est pas comparer ! . . . 23
2.7.4 Les variantes de l'affectation . . . 23
2.7.5 Représentation graphiques des affectations . . . 24
2.7.6 Suppression d'une variable . . . 24
2.8 Les chaînes de caractères . . . 25
2.8.1 Présentation . . . 25
2.8.2 Les séquences d'échappement . . . 25
2.8.3 Opérations . . . 26
2.8.4 Fonctions *vs* méthodes . . . 26
2.8.5 Méthodes de test de l'état d'une chaîne . . . 26
2.8.6 Méthodes retournant une nouvelle chaîne . . . 27
2.8.7 Méthode retournant un indice . . . 27
2.8.8 Indexation simple . . . 28
2.8.9 Extraction de tranches . . . 28
2.8.10 Opérateur de formatage des chaînes . . . 29
2.8.11 Affichage formaté . . . 30
2.9 Les types binaires . . . 32
2.10 Les entrées-sorties . . . 32
2.10.1 Les entrées . . . 32
2.10.2 Les sorties . . . 33
2.11 Exercices . . . 34

3 Contrôle du flux d'instructions 35
3.1 Instructions composées . . . 35
3.2 Choisir . . . 36
3.2.1 Choisir : `if - [elif] - [else]` . . . 36
3.2.2 Syntaxe compacte d'une alternative . . . 37
3.3 Boucles . . . 37
3.3.1 Répéter : `while` . . . 38
3.3.2 Parcourir : `for` . . . 38
3.4 Ruptures de séquences . . . 39
3.4.1 Interrompre une boucle : `break` . . . 39
3.4.2 Court-circuiter une boucle : `continue` . . . 39
3.4.3 Traitement des erreurs : les exceptions . . . 40
3.5 Exercices . . . 41

Copyright © 2017 Dunod.

4 Conteneurs standard **43**
4.1 Séquences . . . 43
4.2 Listes . . . 43
4.2.1 Définition, syntaxe et exemples . . . 44
4.2.2 Initialisations et tests d'appartenance . . . 44
4.2.3 Méthodes modificatrices . . . 44
4.2.4 Manipulation des index et des « tranches » . . . 45
4.3 Tuples . . . 45
4.4 Séquences de séquences . . . 46
4.5 Retour sur les références . . . 47
4.6 Dictionnaires (`dict`), exemples de tableaux associatifs . . . 49
4.7 Ensembles (`set`) . . . 50
4.8 Itérer sur les conteneurs . . . 51
4.9 Exercices . . . 52

5 Fichiers textuels **53**
5.1 Gestion des fichiers . . . 53
5.1.1 Ouverture et fermeture des fichiers . . . 54
5.1.2 Écriture séquentielle . . . 55
5.1.3 Lecture séquentielle . . . 55
5.2 Travailler avec des fichiers et des répertoires . . . 56
5.2.1 Se positionner dans l'arborescence . . . 56
5.2.2 Construction de noms de chemins . . . 56
5.2.3 Opérations sur les noms de chemins . . . 56
5.2.4 Gestion d'un répertoire . . . 57
5.3 Exercices . . . 57

6 Fonctions et espaces de noms **59**
6.1 Définition et syntaxe . . . 59
6.2 Passage des arguments . . . 61
6.2.1 Mécanisme général . . . 61
6.2.2 Un ou plusieurs paramètres, pas de retour . . . 62
6.2.3 Un ou plusieurs paramètres, un ou plusieurs retours . . . 62
6.2.4 Passage d'une fonction en paramètre . . . 63
6.2.5 Paramètres avec valeur par défaut . . . 63
6.2.6 Nombre d'arguments arbitraire : passage d'un tuple de valeurs . . . 64
6.2.7 Nombre d'arguments arbitraire : passage d'un dictionnaire . . . 64
6.2.8 Argument modifiable (ou *mutable*) . . . 65
6.3 Espaces de noms . . . 65
6.3.1 Portée des objets . . . 66
6.3.2 Résolution des noms : règle « LGI » . . . 66
6.4 Exercices . . . 67

7 Modules et packages **69**
7.1 Modules . . . 69
7.1.1 Import . . . 70
7.1.2 Localisation des fichiers modules . . . 71

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

7.1.3 Emplois et chargements des modules . . . 71
7.2 *Batteries included* . . . 77
7.3 Python scientifique . . . 80
7.3.1 Bibliothèques mathématiques et types numériques . . . 81
7.3.2 L'interpréteur IPython . . . 82
7.3.3 La bibliothèque `NumPy` . . . 83
7.3.4 La bibliothèque `matplotlib` . . . 85
7.3.5 La bibliothèque `SymPy` . . . 86
7.4 Bibliothèques tierces . . . 87
7.4.1 Une grande diversité . . . 87
7.4.2 Un exemple : la bibliothèque `Unum` . . . 87
7.5 Packages . . . 88
7.6 Exercices . . . 88

8 La programmation orientée objet 91
8.1 Origine et évolution . . . 91
8.2 Terminologie . . . 92
8.3 Définition des classes et des instanciations d'objets . . . 93
8.3.1 L'instruction `class` . . . 93
8.3.2 L'instanciation et ses attributs . . . 94
8.3.3 Retour sur les espaces de noms . . . 96
8.4 Méthodes . . . 97
8.5 Méthodes spéciales . . . 98
8.5.1 L'initialiseur . . . 98
8.5.2 Surcharge des opérateurs . . . 98
8.5.3 Exemple de surcharge . . . 99
8.6 Héritage et polymorphisme . . . 99
8.6.1 Formalisme de l'héritage et du polymorphisme . . . 100
8.6.2 Exemple d'héritage et de polymorphisme . . . 101
8.7 Notion de « conception orientée objet » . . . 102
8.7.1 Relation, association . . . 102
8.7.2 Dérivation . . . 103
8.8 Exercice . . . 104

9 La programmation graphique orientée objet 107
9.1 Programmes pilotés par des événements . . . 107
9.2 La bibliothèque `tkinter` . . . 107
9.2.1 Présentation . . . 107
9.2.2 Les widgets de `tkinter` . . . 109
9.2.3 Le positionnement des widgets . . . 109
9.3 Deux exemples . . . 110
9.3.1 Une calculette . . . 110
9.3.2 `tkPhone` . . . 110
9.4 Exercices . . . 117

Copyright © 2017 Dunod.

10 Quelques techniques avancées de programmation 119
10.1 Techniques procédurales . . . 119
10.1.1 Le pouvoir de l'introspection . . . 119
10.1.2 Gestionnaire de contexte (ou bloc gardé) . . . 121
10.1.3 Utiliser un dictionnaire pour lancer des fonctions ou des méthodes . . . 122
10.1.4 Les fonctions récursives . . . 122
10.1.5 Les listes définies en compréhension . . . 125
10.1.6 Les dictionnaires définis en compréhension . . . 126
10.1.7 Les ensembles définis en compréhension . . . 126
10.1.8 Les générateurs et les expressions génératrices . . . 126
10.1.9 Fonctions incluses et fermetures . . . 128
10.1.10 Les décorateurs . . . 129
10.2 Techniques objets . . . 131
10.2.1 Les *Functors* . . . 131
10.2.2 Les accesseurs . . . 132
10.2.3 Le *duck typing*... . . . 135
10.2.4 Le *duck typing*... et les annotations . . . 136
10.3 Techniques fonctionnelles . . . 137
10.3.1 Directive `lambda` . . . 137
10.3.2 Les fonctions `map`, `filter` et `reduce` . . . 138
10.3.3 Les applications partielles de fonctions . . . 139
10.3.4 Programmation fonctionnelle *pure* . . . 139
10.4 La persistance et la sérialisation . . . 141
10.4.1 Sérialisation avec `pickle` et `json` . . . 141
10.4.2 Stockage avec `sqlite3` . . . 142
10.5 Les tests . . . 143
10.5.1 Tests unitaires et tests fonctionnels . . . 143
10.5.2 Module `unittest` . . . 144
10.6 La documentation des sources . . . 145
10.6.1 Le format reST . . . 146
10.6.2 Le module `doctest` . . . 148
10.6.3 Le développement dirigé par la documentation . . . 150
10.7 Exercices . . . 151

11 Solutions des exercices 153

Annexes

A Interlude 167

B La distribution Pyzo et Jupyter Notebook 169

C Jeux de caractères et encodage 177

D Les expressions régulières 181

E Les messages d'erreur de l'interpréteur 189

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

F Résumé de la syntaxe 207

Bibliographie et webographie 217

Glossaire et lexique anglais/français 219

Index 231

Pour aller plus loin et mettre toutes les chances de votre côté, des ressources complémentaires sont disponibles sur le site www.dunod.com.

Connectez-vous à la page de l'ouvrage (grâce aux menus déroulants, ou en saisissant le titre, l'auteur ou l'ISBN dans le champ de recherche de la page d'accueil).
Sur la page de l'ouvrage, sous la couverture, cliquez sur le lien « LES + EN LIGNE ».

Copyright © 2017 Dunod.

Avant-propos

En se partageant, le savoir ne se divise pas, il se multiplie.

À qui s'adresse ce livre ?

Issu d'un cours pour les étudiants du département « Mesures physiques » de l'IUT d'Orsay, profondément remanié, ce livre s'adresse en premier lieu aux étudiants débutant en programmation, issus des IUT, des BTS, des licences pro et scientifiques, des écoles d'ingénieurs, aux élèves des classes préparatoires scientifiques, aux enseignants du secondaire (et peut-être à leurs élèves les plus motivés) et plus généralement à tout autodidacte désireux d'apprendre Python en tant que premier langage de programmation.

Prenant la programmation à la base, cet ouvrage se veut pédagogique sans cesser d'être pratique. D'une part en fournissant de très nombreux exemples, des exercices corrigés dans le texte et en ligne, et d'autre part en évitant d'être un catalogue exhaustif sur le langage d'apprentissage en offrant d'abord une introduction aux principaux concepts nécessaires à la programmation et en regroupant les éléments plus techniques liés au langage choisi dans un dernier chapitre.

Pour permettre néanmoins d'approfondir ces détails, il propose plusieurs moyens de navigation : une table des matières détaillée en début et, en fin, des résumés syntaxiques et fonctionnels complets, un glossaire bilingue et un index. De plus, l'ouvrage offre, sur les pages intérieures de la couverture, le mémento Python 3.

Au-delà de l'apprentissage scolaire de la programmation grâce au langage Python, ce livre aborde des aspects souvent négligés, à savoir : le processus de réflexion utilisé dans la phase d'analyse préalable, la façon de passer de l'analyse à l'écriture du programme, de découper proprement celui-ci en fichiers modules réutilisables et ensuite d'utiliser les outils et techniques pour corriger les erreurs.

Nos choix

Cet ouvrage repose sur quelques partis pris :

— la version 3 du langage Python [1] ;

— le choix de logiciels libres [2] : la distribution Python scientifique `Pyzo`, `miniconda3`, `Jupyter Notebook` et sur des outils *open source* de production de documents (XeLaTeX) ;

— une introduction à la programmation, mais aussi à de nombreux à-côtés souvent oubliés et que l'on ne découvre qu'avec l'expérience.

1. Version qui abolit la compatibilité descendante avec les versions antérieures. C'est une grave décision, mûrement réfléchie : « *Un langage qui bouge peu permet une industrie qui bouge beaucoup* » (Bertrand Meyer).

2. Cf. annexe F p. 218.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Les exercices

Outre les exercices proposés à la fin de chaque chapitre [1] et corrigés en fin d'ouvrage, **plus de 100 exercices corrigés supplémentaires** au format *notebook* accompagnent ce cours sur la page web dédiée à l'ouvrage (https://www.dunod.com/sciences-techniques/python-3).

Présentation des codes

Un code Python *interprété* sera présenté sous la forme :

```
>>> import math
>>> print("pi / 2 =", math.pi/2)
pi / 2 = 1.5707963267948966
```

Un *script* complet ou un fragment de script Python sera présenté sous la forme :

```
# -*- coding: utf8 -*-
"""Un exemple de script."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
import math

# Programme principal ========================================================
print("pi / 2 =", math.pi / 2)
```

Les commandes textuelles, résultats d'exécution ou fichier texte seront présentés sous la forme :

```
Une commande ou un fichier "texte".
```

Mise en lumière des éléments importants

Exemple d'encart de définition :

Définition

✍ Une variable est un *identificateur* associé à une valeur. En Python, c'est une *référence d'objet.*

Exemple d'encart de remarque :

Remarque

✓ Passage par affectation : chaque paramètre de la définition de la fonction correspond, *dans l'ordre*, à un argument de l'appel. La correspondance se fait par *affectation* des arguments aux paramètres.

Exemple d'encart de syntaxe :

Syntaxe

✎ Les méthodes spéciales portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement.

Exemple d'encart d'alerte :

Attention

☢ Toutes les instructions au même niveau d'indentation appartiennent au même bloc.

1. Soit simples, soit moins simples (notés ✠), soit encore plus difficiles (notés ✠✠).

Copyright © 2017 Dunod.

Pour joindre les auteurs

Vous pouvez nous adresser vos remarques aux adresses électroniques suivantes :

pycours@kordeo.eu
laurent.pointal@limsi.fr

Remerciements

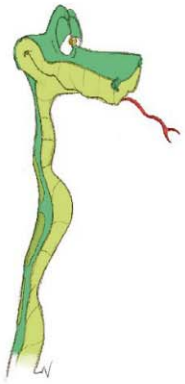
Les auteurs remercient vivement toutes les personnes qui les ont aidés dans la réalisation de ce projet, notamment :

— Hélène Cordeau pour ses illustrations ; les aventures de *Pythoon* enchantent les têtes de paragraphe ! ;
— Jean-Luc Blanc et Brice Martin, des éditions Dunod, qui ont fait un excellent travail critique de relecture ;
— Tarek Ziadé pour les emprunts à son livre (cf. [2], p. 217). Nous remercions les éditions Dunod pour leur aimable autorisation de publier les exemples des pages 144, 146 et 150 ;
— Cécile Trevian pour son aide à la traduction du « Zen de Python » ;
— Xavier Olive pour les emprunts à son article [1] ;
— enfin il faudrait saluer tous les auteurs butinés sur Internet...

Nous tenons également à citer :

— Lucile Roussier, dont la confiance a permis à Laurent de sélectionner et ensuite de promouvoir Python comme langage de script au sein du LURE [2] dès 1995 ;
— nos familles respectives pour avoir supporté nos indisponibilités durant la rédaction de cet ouvrage.

Une pensée spéciale pour Stéphane Barthod : son enseignement didactique auprès des étudiants et son engagement envers ses collègues ne seront pas oubliés.



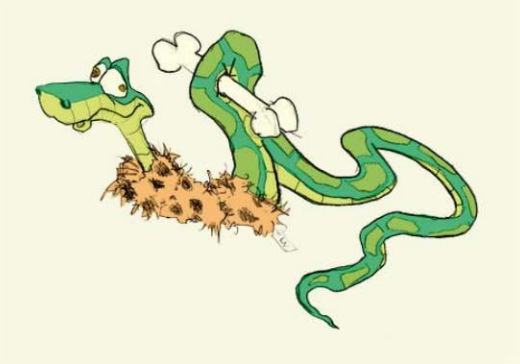
1. http://www.xoolive.org/2014/08/26/introduction-a-lecriture-chinoise.html
2. Laboratoire pour l'Utilisation du Rayonnement Électromagnétique.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Chapitre 1

Programmer en Python



Ce premier chapitre introduit les grandes caractéristiques du langage Python, le replace dans l'histoire des langages informatiques, donne les particularités de production des programmes, définit la notion si importante d'algorithme et conclut sur les diverses implémentations disponibles.

1.1 Mais pourquoi donc apprendre à programmer ?

Avant de plonger dans les détails de la technique, il est important de se demander pourquoi programmer un ordinateur. L'apprentissage des bases de la programmation de logiciels et, corollaire, l'apprentissage des bases de l'algorithmique, permettent de développer des capacités à comprendre la forme séquentielle des processus divers que l'on peut rencontrer dans la vie de tous les jours [1] et à savoir écrire de tels processus d'une façon formelle. Pour certaines personnes à l'esprit plus analytique, plus mathématique, certains concepts et leur mise en œuvre sont évidents et le passage du problème à la programmation de la solution assez facile. Pour d'autres, à l'esprit plus global (on parle de pensée en arborescence ou analogique), le passage du problème à la forme séquentielle de la solution sera moins aisé, l'apprentissage de la programmation et sa pratique peuvent alors apporter le savoir et des techniques leur permettant cette transition.

L'informatique, ou traitement automatique de l'information, est présente un peu partout dans le monde qui nous entoure, et ce de façon de plus en plus envahissante dans la vie de tous les jours (objets « intelligents », smartphones, systèmes de réponse automatique, systèmes auto-pilotés, outils communicants, World-Wide-Web et autre protocoles de l'Internet…).

Avoir des connaissances en programmation permet de démythifier l'aspect « magique » de l'informatique, d'en comprendre les tenants et aboutissants, les risques liés, les impacts sur la vie privée… et dans une certaine limite de s'approprier cette informatique.

1. Une recette de cuisine, une notice de montage ou de réglage, une aide au remplissage d'un formulaire…

Copyright © 2017 Dunod.

Savoir programmer, donc mettre en œuvre pratiquement des algorithmes existants ou écrire ses propres algorithmes, permet d'automatiser des tâches qui seraient pour le moins fortement chronophages ou répétitives et sources d'erreurs potentielles.

L'exemple donné ci-dessous montre comment, à partir d'informations récupérées sur le web, des séries d'opérations permettent d'en extraire celles qui nous intéressent et de générer des graphiques, l'ensemble pouvant être automatisé pour ne plus avoir à les faire « à la main ». Un exemple qui interagirait avec le monde physique demanderait des équipements intermédiaires en entrée et/ou en sortie qui sont au-delà de cet ouvrage [1].

1.1.1 Un exemple pratique

L'exemple présenté ici donne un aperçu de la démarche d'analyse et de programmation, sur un cas proche du réel. Il ne rentre pas dans les détails – le reste de l'ouvrage est là pour ça – mais donne une idée générale de la façon de procéder et de la forme que cela prend.

Pour cet exemple, nous allons donner comme objectif de tracer une cartographie des 200 communes françaises (métropolitaines) ayant la plus faible densité de population.

Pour cela il faut disposer de la liste des villes françaises avec au moins leur localisation et leur densité de population (ou sinon leur surface et leur nombre d'habitants). Après une recherche sur le web, on trouve le site `http://sql.sh/`, dédié à l'apprentissage du langage SQL [2], qui fournit pour ses exemples une liste des villes de France avec une série d'informations [3] dont celles qui nous intéressent [4]. Ces données sont entre autres disponibles dans un fichier `villes_france.csv`, au format `Comma Separated Values` qui nous facilitera la lecture en Python.

Première étape : la lecture

Après avoir récupéré localement le fichier de données (et regardé la description qui en est faite sur le site `sql.sh` pour identifier les colonnes qui nous intéressent), nous allons pouvoir réaliser la première étape pour notre exemple : charger les données des communes en mémoire. Pour cela nous allons utiliser le module Python de manipulation des fichiers `csv` et charger les données dans une liste. Dans cette première partie du script, nous avons aussi ajouté deux filtrages, un premier pour éliminer les communes non métropolitaines (à partir du code postal), et un second pour éliminer les communes dont l'information de population n'est pas disponible (ou qui ne compte aucun habitant). Les données sont converties de leur représentation textuelle issue du fichier CSV vers le format naturel pour leur utilisation.

```
#!/usr/bin/python3
# -*- encoding: utf8 -*-

import csv
from collections import namedtuple
```

1. Cf. le nano-ordinateur monocarte Raspberry Pi `https://www.raspberrypi.org/products/` et l'interpréteur MicroPython `https://micropython.org/`
2. *Structured Query Language*, un langage dédié à la manipulation de données structurées à partir d'une algèbre relationnelle – langage très différent de Python mais qui peut être très utile à apprendre aussi.
3. `https://sql.sh/736-base-donnees-villes-francaises`
4. Sous une licence Creative Common BY SA.

Copyright © 2017 Dunod.

```
Ville = namedtuple("Ville", "nom lat lng nbhab surf dens")

print("===== Lecture des villes de France")
# Source: https://sql.sh/736-base-donnees-villes-francaises
lectcsv = csv.reader(open('villes_france.csv', encoding='utf8'))
villes = []                     # Part d'une liste vide
nbtotvilles = 0                 # Comptage nombre total de villes, pour info
for items in lectcsv:
    nbtotvilles += 1
    # On filtre les départements hors France métropolitaine.
    # Certaines communes ont plusieurs codes postaux, séparés par des
    # tirets dans le fichier.
    cp = items[8].split('-')    # => Codes postaux
    dept = int(cp[0][:2])       # (département à partir du 1er code postal)
    if not 1 <= dept <= 95:
        continue                # Commune ignorée, passe à la suivante

    hab = int(items[16])        # => Population estimée, 2012
    # On filtre les communes dont la population est inconnue (nulle)
    if hab == 0:
        continue                # Commune ignorée, passe à la suivante

    n = items[5]                # => Nom de la commune
    surf = float(items[18])     # => Surface
    lat = float(items[20])      # => Latitude
    lng = float(items[19])      # => Longitude
    v = Ville(n, lat, lng, hab, surf, hab / surf)
    villes.append(v)            # Références ajoutées

print("Nombre de villes considérées:", len(villes), "sur", nbtotvilles)
```

Si on exécute ce programme, on devrait déjà avoir, après une exécution rapide, le résultat suivant sur notre console :

```
===== Lecture des villes de France
Nombre de villes considérées: 35642 sur 36700
```

Deuxième étape : le tri

Nous avons toutes les données nécessaires en mémoire dans une liste. Trier ces données est trivial en Python (et avoir utilisé les `namedtuple` permet d'identifier simplement l'information servant de critère de tri – au passage, établir un tri sur un autre critère est très simple, et une option permet d'effectuer un tri décroissant si on le désire) :

```
print("=== Villes par densité croissante")
villes.sort(key=lambda x: x.dens)   # Tri sur l'attribut dens des communes.
```

Pour un affichage texte indicatif du résultat, nous allons utiliser un module outil de Python qui se charge de faire des « affichages propres » de données structurées, et nous allons nous limiter aux 10 communes les moins densément peuplées. On ajoute au début du script l'import de ce module outil :

```
from pprint import pprint
```

Et après l'instruction de tri des villes, on demande l'affichage des 10 premières :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
pprint(villes[:10])
```

Ce qui à l'exécution nous ajoute ces affichages sur notre console :

```
=== Villes par densité croissante
[Ville(nom='Saint-Christophe-en-Oisans', lat=44.9667, lng=6.18333, nbhab=100, surf=123.47,
dens=0.8099133392726978),
 Ville(nom='Asco', lat=42.4537, lng=9.03251, nbhab=100, surf=122.81, dens=0.814265939255761),
 Ville(nom='Manso', lat=42.3659, lng=8.79251, nbhab=100, surf=121.02, dens=0.8263097008758883),
 Ville(nom='La Chapelle-en-Valgaudémar', lat=44.817, lng=6.19473, nbhab=100, surf=108.02,
dens=0.9257544899092761),
 Ville(nom='Saint-Paul-sur-Ubaye', lat=44.515, lng=6.75167, nbhab=200, surf=205.55,
dens=0.9729992702505472),
 Ville(nom='Champoléon', lat=44.7333, lng=6.23333, nbhab=100, surf=98.54,
dens=1.0148163182463974),
 Ville(nom='Callen', lat=44.3, lng=-0.466667, nbhab=100, surf=87.86, dens=1.1381743683132255),
 Ville(nom='Villiers-le-Duc', lat=47.8167, lng=4.71667, nbhab=100, surf=84.34,
dens=1.1856770215793218),
 Ville(nom='Prads-Haute-Bléone', lat=44.22, lng=6.44334, nbhab=200, surf=165.64,
dens=1.2074378169524271),
 Ville(nom='Gavarnie', lat=42.7333, lng=-0.008333, nbhab=100, surf=82.54,
dens=1.21153380179307)]
```

Troisième étape : le dessin de la carte

On pourrait se lancer dans la recherche des coordonnées latitude/longitude des littoraux et des frontières, trouver un module de dessin, faire les conversions entre ces coordonnées et les pixels... mais cela serait bien se compliquer la vie. Python dispose, parmi les outils scientifiques que l'on peut installer en supplément, d'un bon module de tracé de courbes, agrémenté d'une partie dédiée au tracé de cartes. Pour cela, à partir de l'environnement de développement que nous avons choisi, nous installons `matplotlib, mpl_toolkits` et son `Basemap` [1].

Pour démarrer, rien de tel qu'un exemple déjà fonctionnel que l'on va modifier et adapter pour répondre à nos besoins, éventuellement en piochant des éléments d'autres exemples trouvés ailleurs. `matplotlib` dispose pour cela d'une galerie d'exemples [2] très bien fournie. Et pour `Basemap` il existe un tutoriel [3] ainsi que de nombreux exemples en ligne.

Après avoir réussi à centrer la carte sur la France et à fixer les limites d'affichage pour avoir la métropole en incluant la Corse avec une projection le permettant (nombreux essais/corrections), on peut jouer avec les couleurs et les épaisseurs de traits, affiner l'affichage des méridiens et parallèles... On arrive au résultat suivant :

```
#!/usr/bin/python3
# -*- encoding: utf8 -*-

from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

# Ces deux lignes évitent des messages d'alerte concernant des fonctions de
# matplotlib en cours de dépréciation et encore utilisées par mpl_toolkit
```

1. `conda install numpy matplotlib basemap`
2. https://matplotlib.org/gallery.html
3. http://basemaptutorial.readthedocs.io/en/latest/plotting_data.html

Copyright © 2017 Dunod.

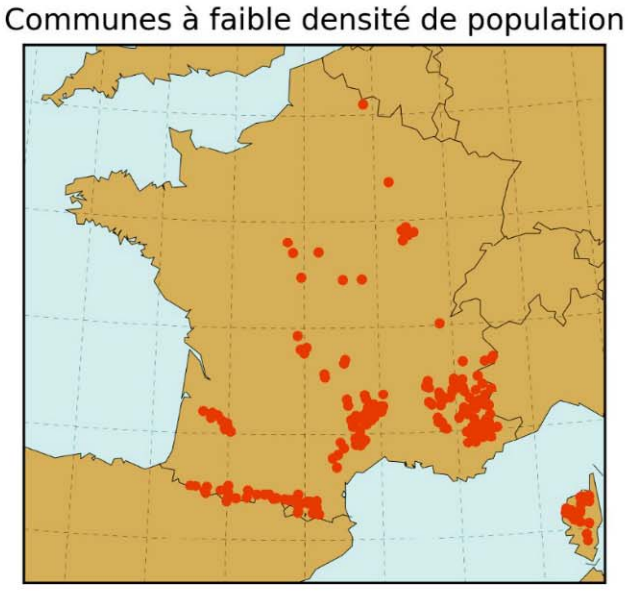
```
# (à la date d'écriture du script) - vous pouvez les commenter pour voir ces messages
import warnings
warnings.filterwarnings("ignore")

# Choix de la projection, du centrage et de la mise à l'échelle
carte = Basemap(projection='stere', lat_0=46.60611, lon_0=1.87528,
    resolution='l', llcrnrlon=-5, urcrnrlon=11, llcrnrlat=41, urcrnrlat=51)
# Tracé des lignes de cotes, pays. Couleur pour les continents/la mer
carte.drawcoastlines(linewidth=0.25)
carte.drawcountries(linewidth=0.25)
carte.fillcontinents(color='#CAAF68', lake_color='#D3FFFF')
carte.drawmapboundary(fill_color='#D3FFFF')
# Lignes parallèles/méridiens tous les 2 degrés
carte.drawmeridians(np.arange(0, 360, 2), linewidth=0.1)
carte.drawparallels(np.arange(-90, 90, 2), linewidth=0.1)
plt.title('Communes à faible densité de population')
```

Dernière étape : tracer les villes

Il nous reste à ajouter des points correspondant aux 200 communes qui nous intéressent... La façon de tracer un point se trouve aisément dans les exemples, et le faire pour 200 communes est simple.

Enfin on enregistre et on affiche la carte.

```
for v in villes[:200]:
    # Passage de coordonnées lat/long en coordonnées sur le graphique.
    x, y = map(v.lng, v.lat)
    map.plot(x, y, marker='o', color='Red', markersize=3)

# Si on veut sauvegarder la figure
plt.savefig("figure.png", dpi=300)  # sinon on commente la ligne
plt.show()                          # affichage de la carte de la France dans une fenêtre
```



Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Le code final

Le code final (`carto.py`) est téléchargeable sur le site de Dunod : https://www.dunod.com/sciences-techniques/python-3.

1.1.2 Et après ?

De l'apprentissage de l'algorithmique et des bases de la programmation au métier d'informaticien et à ses nombreuses branches, il y a bien des aspects à approfondir : la structuration des données et les algorithmes associés, les aspects matériels (éventuellement le traitement du signal si l'on s'intéresse aux transmissions des signaux dans les réseaux), le fonctionnement des systèmes informatiques sur ces matériels, beaucoup de normes et protocoles standard sur l'information et sa représentation, l'ingénierie autour de la création de logiciel et de l'écriture de code, les outils de travail collaboratif, les outils de sécurité et de chiffrement... Dans le temps et avec l'expérience, viennent aussi la connaissance de librairies tierces et de leur utilisation, le recul par rapport à ce que l'on écrit, la mise en œuvre de bonnes pratiques, etc.

En parallèle à ces aspects plutôt techniques, il y a aussi des aspects plus humains : un côté artisan travaillant sur son ouvrage, un côté relationnel lorsque l'on est avec les utilisateurs en phase amont pour comprendre leurs besoins et ensuite pour corriger les erreurs et faire évoluer le logiciel, du développement en collaboration lorsque le projet est de taille importante, parfois des côtés esthétiques et/ou ergonomiques à prendre en compte, etc.

Mais il n'est nul besoin d'être un informaticien de métier pour pouvoir déjà créer des outils informatiques adaptés à ses propres besoins.

1.2 Mais pourquoi donc apprendre Python ?

Python est un langage facile à apprendre, son code est réputé concis et clair. C'est un langage à usage général, multi-plateforme et *open source*. Il est accompagné d'une importante bibliothèque standard. De plus, il dispose de `PyPi`, un gestionnaire de paquets de plus de 85 000 modules qui offrent des solutions pour résoudre des problèmes dans des domaines très divers.

Python est un bon choix pour apprendre la programmation car sa syntaxe est très proche d'une notation algorithmique, base de la programmation. Ce livre est un terme moyen entre une présentation attrayante mais superficielle et une présentation formelle basée sur l'algorithmique pure et dure.

Une dizaine d'années d'enseignement en première année universitaire nous ont convaincus de l'intérêt d'une démarche progressive ponctuée d'exemples (les outils modernes comme `Jupyter Notebook` renforcent cette démarche) et de la présentation de l'étude des fonctions avant celle des classes. Les deux démarches, fonctionnelle et orientée objet, sont certes importantes et formatrices mais, même si le paradigme objet se conçoit bien en Python et s'énonce clairement, le développement modulaire de fonctions permet déjà de satisfaire rapidement de nombreux besoins.

Copyright © 2017 Dunod.

1.2.1 Principales caractéristiques du langage Python

Ce chapitre se contente de citer les points forts du langage, caractéristiques qui seront développées tout au long de l'ouvrage.

Historique :

- 1991 : Guido van Rossum travaille aux Pays-Bas [1] sur le projet AMOEBA, un système d'exploitation distribué. Il conçoit Python à partir du langage ABC et publie la version 0.9.0 sur un forum Usenet,
- 1996 : sortie de *Numerical Python*, ancêtre de *numpy*,
- 1999 : premières journées Python-France à l'Onera [2],
- 2001 : naissance de la PSF (Python Software Fundation),
- les versions se succèdent… Un grand choix de modules est disponible, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise…,
- 2006 : première sortie de IPython,
- fin 2008 : sorties simultanées de Python 2.6 et de Python 3.0,
- 2017 : versions en cours des deux branches 2.7.13 et 3.6.1.

Langage open source :

- licence open source CNRI, compatible GPL, mais sans la restriction *copyleft*. Donc Python est libre et gratuit même pour les usages commerciaux,
- pilotage des évolutions du langage par la communauté des utilisateurs, *via* les PEP [3]
- GvR (Guido van Rossum) est le « BDFL » (dictateur bénévole à vie !),
- importante communauté de développeurs,
- nombreux outils standard disponibles : Python est fourni *batteries included* (avec les piles).

Travail interactif :

- nombreux environnements interactifs disponibles (notamment Jupyter),
- importantes documentations en ligne,
- développement rapide et incrémentiel,
- tests et débogage outillés,
- analyse interactive de données.

Langage interprété rapide :

- interprétation du *bytecode* compilé,
- de nombreux modules sont disponibles à partir de bibliothèques optimisées (souvent écrites en C ou C++).

Simplicité du langage (cf. annexe A p. 167) :

- syntaxe claire et cohérente : notations identiques reprises pour les utilisations similaires,
- indentation significative : la forme reflète visuellement la structure,
- gestion automatique de la mémoire (*garbage collector*),
- typage dynamique fort : pas de déclaration (mais pas de mutation « magique » !).

Orientation objet :

- modèle objet puissant mais pas obligatoire,
- structuration multifichier aisée des applications, ce qui facilite les modifications et les extensions,

1. Au CWI : Centrum voor Wiskunde en Informatica.
2. Office National d'Études et de Recherches Aérospatiales.
3. Python Enhancement Proposals, https://www.python.org/dev/peps/

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

— les classes, les fonctions et les méthodes sont des objets dits *de première classe*. Ces objets sont traités comme tous les autres (on peut les affecter, les passer en paramètre).

Ouverture au monde :

— interfaçable avec C/C++/FORTRAN...,
— langage de script utilisé dans de nombreuses applications dans le monde,
— excellente portabilité.

Disponibilité de bibliothèques :

— plusieurs milliers de packages sont disponibles dans tous les domaines.

On définit parfois Python comme un *langage algorithmique* exécutable

1.2.2 Implémentations de Python

Remarque

✓ Une « implémentation » signifie une « mise en œuvre ».

— **CPython** : *Classic Python*, codé en C, implémentation portable sur différents systèmes, la référence du langage.
— **MicroPython** : version optimisée et allégée de Python 3 pour système embarqué. Cf. par exemple le site `http://wiki.mchobby.be/index.php?title=MicroPython-Accueil`.
— **Jython** : ciblé pour la JVM (Java Virtual Machine).
— **IronPython** : *Python.NET*, écrit en C#, utilise le MSIL (*MicroSoft Intermediate Language*).
— **Stackless Python** : élimine l'utilisation de la pile du langage C (permet de récurser [1] tant que l'on veut).
— **Pypy** : projet de recherche européen d'un interpréteur Python écrit en une version restreinte de Python.

Dans cet ouvrage, nous utiliserons CPython, l'implémentation de référence du langage.

1.3 Comment passer du problème au programme

Au fur et à mesure que l'on acquiert de l'expérience, on découvre et on apprend à utiliser les bibliothèques de modules et paquets qui fournissent des types de données et des services avancés, évitant d'avoir à recréer, coder et déboguer une partie de la solution. C'est la méthode que nous avons employée dans l'exemple pratique du début de ce chapitre.

Lorsqu'on a un problème à résoudre par un programme, la difficulté est de savoir :

1. par où commencer ;
2. comment concevoir l'algorithme.

1.3.1 Réutiliser

La première chose à faire est de vérifier qu'il n'existe pas déjà une solution (même partielle) au problème que l'on pourrait reprendre *in extenso* ou dont on pourrait s'inspirer. On peut chercher dans les nombreux modules standard installés avec le langage, dans les dépôts institutionnels de modules tiers (le *Python Package Index* [2] par exemple), ou encore utiliser les moteurs de recherche

1. La notion de récursion est détaillée cf. § 10.1.4 p. 122.
2. `http://pypi.python.org/`

Copyright © 2017 Dunod.

sur l'Internet. Si on ne trouve pas de solution existante dans notre langage préféré, on peut trouver une solution dans un autre langage, qu'il n'y aura « plus qu'à » adapter.

1.3.2 Réfléchir à un algorithme

L'analyse qui permet de créer un algorithme et la programmation ensuite sont deux phases qui nécessitent de la pratique avant de devenir « évidentes » pour des problèmes faciles.

Définition

✍ **Algorithme** : ensemble des étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions. Donc un algorithme se termine en un *temps fini.*

Pour démarrer, il faut partir d'éléments réels, mais sur un échantillon du problème comportant peu de données, un cas que l'on est capable de traiter « à la main ».

Il est fortement conseillé de démarrer sur papier ou sur un tableau (le papier ayant l'avantage de laisser plus facilement des traces des différentes étapes).

On identifie tout d'abord quelles sont les données que l'on a à traiter en entrée et quelles sont les données que l'on s'attend à trouver en sortie. Pour chaque donnée, on essaie de préciser quel est son domaine, quelles sont ses limites, quelles contraintes la lient aux autres données.

1.3.3 Résoudre « à la main »

On commence par une résolution du problème, en réalisant les transformations et calculs sur notre échantillon de problème, en fonctionnant par étapes.

À chaque stade, on note :

— quelles sont les étapes pertinentes, sur quels critères elles ont été choisies ;
— quelles sont les séquences d'opérations que l'on a répétées.

Lorsque l'on tombe sur des étapes complexes, on découpe en sous-étapes, éventuellement en les traitant séparément comme un algorithme de résolution d'un sous-problème. Le but est d'arriver à un niveau de détail suffisamment simple ; soit qu'il s'agisse d'opérations très basiques (opération sur un texte, expression de calcul numérique…), soit que l'on pense/sache qu'il existe déjà un outil pour traiter ce sous-problème (calcul de sinus pour un angle, opération de tri sur une séquence de données…).

Lors de ce découpage, il faut éviter de considérer des opérations comme « implicites » ou « évidentes », il faut préciser d'où proviennent les informations et ce que l'on fait des résultats. Par exemple, on ne considère pas « un élément » mais « le nom traité est l'élément suivant de la séquence de noms » ou encore « le nom traité est le x^e élément de la séquence de noms ».

Normalement, au cours de ces opérations, on a commencé à nommer les données et les étapes au fur et à mesure qu'on en a eu besoin.

1.3.4 Formaliser

Une fois qu'on a un brouillon des étapes, il faut commencer à mettre en forme et à identifier les constructions algorithmiques connues et les données manipulées :

— répétitions de traitement (sur quelles informations ?, condition d'arrêt) ;
— tests (quelles conditions ?) ;

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

- informations en entrée, quel est leur type ? quelles sont les contraintes pour qu'elles soient valides et utilisables ? d'où viennent-elles ? :
  - déjà présentes en mémoire,
  - demandées à l'utilisateur,
  - lues dans des fichiers ou récupérées ailleurs (sur l'Internet par exemple), dans quel format ? ;
- calculs et expressions :
  - quel genre de données sont nécessaires ? y a-t-il des éléments constants à connaître, des résultats intermédiaires à réutiliser ?,
  - on peut identifier ici les contrôles intermédiaires possibles sur les valeurs qui puissent permettre de vérifier que l'algorithme se déroule bien ;
- stockage des résultats intermédiaires ;
- résultat final — à quel moment l'obtient-on ? qu'en fait-on ? :
  - retourné dans le cadre d'une fonction,
  - affiché à l'utilisateur,
  - sauvegardé dans un fichier.

1.3.5 Factoriser

Le but est d'identifier les séquences d'étapes qui se répètent en différents endroits, séquences qui seront de bons candidats pour devenir des fonctions ou des méthodes de classes. Ceci peut être fait en même temps que l'on formalise.

1.3.6 Passer de l'idée au programme

Définition

✍ **Programme** : un programme est une *traduction d'un algorithme* en un langage compilable ou interprétable par un ordinateur. Il est souvent écrit en plusieurs parties dont une qui *pilote* les autres : le *programme principal.*

Le passage de l'idée puis de l'algorithme au code dans un programme est relativement facile en Python car celui-ci est très proche d'un langage d'algorithmique.

- Les ***noms*** des choses que l'on a manipulées vont nous donner des **`variables`**.
- Les ***tests*** vont se transformer en **`if condition:`**
- Les ***répétitions***[1] ***sur des séquences*** d'informations vont se transformer en **`for variable in séquence:`**
- Les ***répétitions avec expression*** de condition vont se transformer en **`while conditions:`**
- Les ***séquences d'instructions qui apparaissent en différents endroits*** vont se transformer en **`fonctions`**.
- Le ***retour de résultat*** d'une séquence (fonction) va se traduire en **`return variable`**.
- Les ***conditions sur les données*** nécessaires pour un traitement vont identifier des tests d'erreurs et des levées d'exception.

1. On parlera de *boucles.*

Copyright © 2017 Dunod.

1.4 Environnements matériel et logiciel

1.4.1 L'ordinateur

Définition

✍ On peut schématiser la définition de l'**ordinateur** de la façon suivante : **automate fini déterministe à composants électroniques**. Un tel automate est composé d'un ensemble d'états ; le passage d'un état actif à un autre est réalisé de façon unique par une transition déterminée par les données.

L'ordinateur comprend en substance :

- un **microprocesseur**, une horloge, une mémoire cache rapide ;
- de la **mémoire volatile** (dite *vive* ou RAM), contenant les instructions et les données nécessaires à l'exécution des programmes. La RAM est formée de cellules binaires (*bits* avec deux états 0/1) organisées en mots de 8 bits [1] (*octets*) ;
- des **périphériques** : entrées/sorties, mémoires permanentes [2] (disque dur, clé USB, CD-ROM…), réseau…

1.4.2 Deux sortes de programmes

On distingue, pour faire rapide :

- le **système d'exploitation** : ensemble des programmes qui gèrent les ressources matérielles et logicielles. Il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interpréteur de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;
- les **programmes applicatifs** sont dédiés à des tâches particulières. Ils sont formés d'une série d'instructions contenues dans un programme *source* lisible par un humain qui est traduit en un programme exécutable par l'ordinateur.

1.5 Différents niveaux de langages

Chaque microprocesseur possède un langage propre, directement exécutable : le *langage machine* binaire. Ce langage n'est pas portable [3], c'est le *seul* que l'ordinateur puisse utiliser.

Le *langage d'assemblage* est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine mais n'est toujours pas portable. On le traduit en langage machine en utilisant un programme assembleur.

Les *langages de haut niveau*. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits dans le langage machine adapté au microprocesseur par un *compilateur* ou un *interpréteur*.

Des milliers de langages de programmation ont été créés, d'autres continuent d'apparaître, mais l'industrie n'en utilise qu'une minorité.

1. Pour la grande majorité des cas, mais d'autres organisations ont été utilisées.
2. Les données y persistent lorsque l'ordinateur n'est plus alimenté.
3. Il n'est pas exécutable par un microprocesseur d'une autre famille.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

1.6 Les techniques de production des programmes

La *compilation* est la traduction du texte du programme (dit *source*) en une représentation quasi prête pour le microprocesseur (dit *objet*, mais qui n'a rien à voir avec la « programmation objet »). Elle comprend au moins quatre phases (trois phases d'analyse – lexicale, syntaxique et sémantique – et une phase de production de code *objet*). Pour générer le langage machine il faut encore une phase particulière : l'*édition de liens* à partir du code *objet*. La compilation est contraignante mais offre au final une grande vitesse d'exécution du programme.

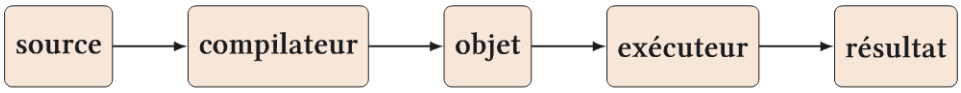


FIGURE 1.1 – Chaîne de compilation

Dans la technique de l'*interprétation*, chaque ligne du *source* analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. Cette technique est très souple, mais les codes générés sont peu performants : l'interprétation doit être réalisée à chaque nouvelle exécution…

FIGURE 1.2 – Chaîne d'interprétation

1.6.1 Technique de production de Python

Le concepteur de Python a opté pour une technique mixte : l'*interprétation du bytecode compilé*, bon compromis entre la facilité de développement et la rapidité d'exécution. Le *bytecode* (forme de représentation intermédiaire du programme) est portable sur tout ordinateur muni de la *machine virtuelle Python*.

FIGURE 1.3 – Interprétation du bytecode compilé

Pour **exécuter un programme**, Python charge le fichier source (`.py`) en mémoire vive, en fait l'analyse (lexicale, syntaxique et sémantique), produit le bytecode et enfin l'exécute.

Afin de ne pas refaire inutilement les phases d'analyse et de production, Python sauvegarde le bytecode produit (dans un fichier `.pyo` ou `.pyc`) et recharge simplement le fichier bytecode s'il est plus récent que le fichier source dont il est issu.

En pratique, il n'est pas nécessaire de compiler explicitement une bibliothèque de code, Python gère ce mécanisme de façon transparente.

Note : Python permet toutefois de développer aussi des bibliothèques de code en langage C, qui sont compilées en langage machine et sont donc exécutées directement au niveau du processeur, sans la phase d'interprétation intermédiaire de bytecode.

Copyright © 2017 Dunod.

1.6.2 Construction des programmes

Le génie logiciel étudie les méthodes de construction des programmes. Plusieurs modèles sont envisageables, entre autres :

- la méthodologie **procédurale**. On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous-algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie **objet**. Centrée sur les données, elle est considérée plus stable dans le temps et meilleure dans sa conception. On conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage*) de classes de base dans une construction hiérarchique.

Python offre les *deux* techniques, que l'on peut mélanger suivant les besoins et suivant ce qui apparaît comme le plus adapté au problème à résoudre. Il arrive parfois qu'on construise une solution sous forme procédurale pour ensuite la restructurer partiellement sous une forme objet plus pérenne et plus réutilisable.

1.7 Thèmes de réflexion

Pour ce premier chapitre, nous proposons deux thèmes de réflexion (sans correction) qui vous permettront de mettre votre expérience personnelle en contexte.

1. Essayez d'identifier des tâches pour lesquelles vous avez eu à utiliser une procédure écrite.

2. Dans votre quotidien, y aurait-il une série d'actions répétées que vous pourriez formaliser en une procédure ? Essayez de le faire.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Chapitre 2

La calculatrice Python

Comme tout langage de programmation, Python permet de manipuler des données grâce à un *vocabulaire* de mots réservés et grâce à des *types de données*.

Ce chapitre présente les règles de construction des identificateurs, les types de données simples (les conteneurs seront examinés au chapitre 4) ainsi que les types chaîne de caractères.

Enfin, *last but not least*, ce chapitre s'étend sur les notions non triviales de variable, de référence d'objet et d'affectation.

2.1 Les deux modes d'exécution d'un code Python

Le mode le plus direct et le plus intuitif d'exécution de Python est l'utilisation interactive de l'*interpréteur*. C'est le *shell Python* [1], ou le *mode interprété*. Lorsqu'on tape la commande `python` dans une console, une invite apparaît. L'interpréteur attend vos instructions, les exécute quand vous avez tapé sur la touche Entrée. C'est ce qu'on appelle la *boucle d'évaluation* [2].

```
>>> 5 + 3      Python affiche l'invite. L'utilisateur tape une expression.
8              Python évalue et affiche le résultat...
>>>            ... puis réaffiche l'invite.
```

Mais dès que l'on travaille avec plus que quelques lignes de code, le mode interprété devient malcommode. On passe en *mode script* : on enregistre un ensemble d'instructions Python dans un fichier [3] grâce à un éditeur. Ce script est exécuté ultérieurement (et autant de fois que l'on veut) par une commande ou par une touche du menu de l'éditeur, il peut être corrigé puis réexécuté dans son ensemble.

1. On parle aussi de « console Python ».
2. En anglais *REPL* (*Read-Eval-Print Loop*).
3. On parle alors d'un *script Python*.

Copyright © 2017 Dunod.

2.2 Les commentaires

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement *présenté* et *commenté* de façon pertinente.

La signification de parties non triviales [1] doit être donnée par un *commentaire*. En Python, un commentaire commence par le caractère `#` et s'étend jusqu'à la fin de la ligne :

```
# ~~~~~~~~~~~~~~~~~~~~
# Voici un commentaire
# ~~~~~~~~~~~~~~~~~~~~

9 + 2      # en voici un autre
```

2.3 Identificateurs et mots clés

2.3.1 Identificateurs

Comme tout langage, Python utilise des *identificateurs* pour nommer tout ce qui est manipulé.

Définition

✍ Un **identificateur** Python est une suite non vide de caractères, de longueur quelconque, formée d'**un** *caractère de début* (n'importe quelle lettre Unicode ou le caractère souligné) et de **zéro** (au sens de « aucun ») **ou plusieurs** *caractères de continuation* (lettre Unicode, caractère souligné ou chiffre).

Attention

☢ Les identificateurs sont sensibles à la casse et ne doivent pas faire partie des mots réservés de Python 3 (cf. § 2.3.3 p. 17).

Le choix d'un bon identificateur est important car il doit permettre, lors de la rédaction et de la lecture du code, de comprendre ce qu'il représente ; c'est un point qu'il ne faut pas négliger.

On peut avoir des identificateurs très courts comme `x, y, z, a, b, c`... pour autant que leur sens dans le contexte soit pertinent (`x` pour une valeur de calcul ; `a, b, c` pour des coefficients d'une équation ; `f` pour une fonction quelconque...) – on évitera de les utiliser par facilité si cela n'a pas de sens, tout comme des `var1, var2, var3`... pour lesquels il devient difficile de mémoriser l'utilisation au bout de quelques lignes de programme, et qui sont de ce fait souvent causes d'erreurs.

En général quelques caractères permettent déjà de donner du sens comme `som` (somme), `fct` (fonction), `maxi, mini, stop, start`. Mais il ne faut pas hésiter à les allonger si nécessaire : `maxi_x, augmentation_son`...

2.3.2 Style de nommage

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Voici le style utilisé dans le présent ouvrage [2] :

- `NOM_DE_MA_CONSTANTE` pour les constantes ;
- `maFonction`, `maMethode` pour les fonctions et les méthodes ;
- `MaClasse` pour les classes ;

1. Et uniquement celles-ci, un script *bavard* est désagréable !
2. Signalons que certains de nos exemples s'éloignent du style préconisé et suivent les usages de la PEP 8 : « Style Guide for Python », Guido van Rossum et Barry Warsaw. https://www.python.org/dev/peps/pep-0008/

Copyright © 2017 Dunod.

— `UneExceptionError` pour les exceptions ;
— `nom_de_ma_variable` pour les variables et pour tous les autres identificateurs.

Exemples :

```
NB_ITEMS = 12           # appelé "UPPER_CASE_WITH_UNDERSCORES"
class MaClasse: pass    # appelé "CamelCase" ou "CapitalizedWords"
def maFonction(): pass  # appelé "mixedCase"
mon_id = 5              # appelé "lower_case_with_underscores"
```

Pour ne pas prêter à confusion, éviter d'utiliser les caractères `l` (minuscule), `O` et `I` (majuscules) seuls. Enfin, on évitera d'utiliser les notations suivantes :

```
_xxx     # usage interne
__xxx    # attribut de classe
__xxx__  # nom spécial réservé
```

2.3.3 Les mots réservés de Python 3

La version 3.6 de Python compte 33 mots clés :

`and`	`del`	`from`	`None`	`True`
`as`	`elif`	`global`	`nonlocal`	`try`
`assert`	`else`	`if`	`not`	`while`
`break`	`except`	`import`	`or`	`with`
`class`	`False`	`in`	`pass`	`yield`
`continue`	`finally`	`is`	`raise`	
`def`	`for`	`lambda`	`return`	

2.4 Notion d'expression

Définition

✍ Une **expression** est une portion de code que l'interpréteur Python peut **évaluer** pour obtenir une **valeur**.

Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux (représentant directement des valeurs), d'identificateurs et d'opérateurs.

Par exemple :

```
>>> id1 = 15.3
>>> id2 = 4 + 3 * sin(pi)
>>> id3 = "-"*10 + "Titre" + "-"*10
```

2.5 Les types de données entiers

Définition

✍ Le **type** d'un objet Python détermine de quelle sorte d'objet il s'agit.
La fonction `type()` fournit le type d'une valeur.

Python offre deux types entiers standard : `int` et `bool`.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2.5.1 Le type `int`

Le type `int` n'est limité en taille que par la mémoire de la machine [1].

Les entiers littéraux sont représentés en décimal par défaut, mais on peut aussi utiliser les bases suivantes :

```
>>> 2013            # décimal (base 10) par défaut
2013
>>> 0b11111011101   # binaire (base 2) avec le préfixe 0b
2013
>>> 0o3735          # octal (base 8) avec le préfixe 0o
2013
>>> 0x7dd           # hexadecimal (base 16) avec le préfixe 0x
2013
>>> # représentations binaire, octale et hexadécimale de l'entier 179
>>> bin(179), oct(179), hex(179)
('0b10110011', '0o263', '0xb3')
```

Ces dernières opérations correspondent à des *changements de bases* classiques [cf. p. 19].

Opérations arithmétiques

Les principales opérations [2] :

```
>>> 20 + 3
23
>>> 20 - 3
17
>>> 20 * 3
60
>>> 20 ** 3
8000
>>> 20 / 3
6.666666666666667
>>> 20 // 3       # division entière
6
>>> 20 % 3        # modulo (reste de la division entière)
2
>>> divmod(20, 3) # division entière et modulo (reste)
(6, 2)
>>> abs(3 - 20)   # valeur absolue
17
```

Bien remarquer le rôle des deux opérateurs de division :

`/` : produit une division flottante, même entre deux entiers [3] ;

`//` : produit une division entière.

Remarquons que la fonction prédéfinie `divmod()` qui prend deux entiers et renvoie la paire `q, r` où `q` est le quotient et `r` le reste de leur division, évite d'utiliser `//` pour obtenir `q` et `%` pour `r`.

1. Dans la plupart des autres langages, les entiers sont codés sur un nombre fixe de bits et ont un domaine de définition limité auquel il convient de faire attention. Par exemple, un entier signé sur 16 bits représente un nombre entre −32 768 et +32 767.
2. Les opérateurs Python sont régis par des règles de priorité (cf. § F p. 207).
3. Ceci est une différence majeure avec de nombreux autres langages (y compris avec Python 2) où une division entre deux entiers est une division obligatoirement entière.

Copyright © 2017 Dunod.

Les bases arithmétiques

Certaines bases sont couramment employées : la base 2 (système binaire) en électronique numérique, les base 8 et 16 (système octal et hexadécimal) en informatique, la base 60 (système sexagésimal) dans la mesure du temps et des angles.

Définition

En arithmétique, une **base** n désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre.
Par exemple, en base n : $57_n = (5 \times n^1) + (7 \times n^0)$

Puisqu'un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base, on peut facilement effectuer un *changement de base* :

$$57_{16} = (5 \times 16^1) + (7 \times 16^0) = 87_{10}$$
$$28_{10} = (3 \times 8^1) + (4 \times 8^0) = 34_8$$

Inversement, dans le deuxième exemple, pour passer de la base 10 à la base n, il faut connaître les puissances successives de n.

2.5.2 Le type `bool`

Les opérateurs booléens de base

En Python les représentations littérales des valeurs booléennes sont notées `False` et `True`. Les opérateurs de base sont notés respectivement `not` (non), `and` (et) et `or` (ou).

Opérateur unaire `not`

a	not(a)
False	True
True	False

Opérateurs binaires `or` et `and`

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Voici les principales caractéristiques du type `bool` [1] :

- deux valeurs possibles : `False` (faux), `True` (vrai) ;
- conversion automatique (ou « transtypage », en anglais *cast*) des valeurs des autres types vers le type booléen : zéro (quel que soit le type numérique), les chaînes et conteneurs [2] vides, la constante `None`, les objets dont une méthode spéciale [3] `__bool__()` ou `__len__()` retourne 0 ou faux sont convertis en booléen `False` ; toutes les autres valeurs sont converties en booléen `True` ;
- opérateurs de comparaison entre deux valeurs comparables, produisant un résultat de type `bool` : `==` (égalité), `!=` (différence), `>`, `>=`, `<` et `<=`

1. Nommé d'après George Boole, logicien et mathématicien britannique du XIXe siècle.
2. Liste, tuple, dictionnaire et ensemble [cf. p. 43].
3. Cf. p. 98.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
>>> 2 > 8
False
>>> 2 <= 8 < 15
True
```

— opérateurs logiques : `not`, `or` et `and`. En observant les tables de vérité des opérateurs `and` et `or` (cf. l'encadré précédent), on remarque que :
  — dès qu'un premier membre a la valeur `False`, l'expression `False and expression2` vaudra `False`. On n'a donc pas besoin d'évaluer `expression2`,
  — de même dès qu'un premier membre a la valeur `True`, l'expression `True or expression2` vaudra `True`, quelque soit la valeur de `expression2` ;

  Cette optimisation est appelée « principe du *shortcut* » ou évaluation « au plus court », elle est automatiquement mise en œuvre par Python lors de l'évaluation des expressions booléennes :

```
>>> (3 == 3) or (9 > 24)
True
>>> (9 > 24) and (3 == 3)
False
```

Attention

☢ Pour être sûr d'avoir un résultat booléen avec une expression reposant sur des valeurs *transtypées*, appliquez `bool()` sur l'expression. En effet, lorsqu'il rencontre des valeurs non booléennes dans une expression logique, Python effectue des conversions de type automatique sur les données. Mais le résultat de l'évaluation utilise les valeurs d'origine :

```
>>> 'a' or False      # 'a' est automatiquement transtypé en booléen
'a'
>>> 0 or 56           # 0 et 56 sont automatiquement transtypés en booléen
56
>>> b = 0
>>> b and 3>2         # b est automatiquement transtypé en booléen
0
```

2.6 Les types de données flottants

Remarque

✓ La notion mathématique de *réel* est une notion idéale. Ce graal est impossible à atteindre en informatique. On utilise une représentation interne (normalisée) permettant de déplacer la virgule grâce à une valeur d'exposant variable. On nommera ces nombres des *nombres à virgule flottante*, ou, pour faire plus court, des *flottants*.

2.6.1 Le type `float`

— Un `float` est noté avec un point décimal (jamais avec une virgule) ou, en notation exponentielle, avec un `e` ou un `E` symbolisant le « 10 puissance » suivi des chiffres de l'exposant. Par exemple : `2.718, .02, 3E10, -1.6e-19, 6.023E23`.
— Les flottants supportent les mêmes opérations que les entiers.
— Ils ont une précision limitée.
— L'import du module standard `math` autorise toutes les opérations mathématiques usuelles. Par exemple :

Copyright © 2017 Dunod.

```
>>> import math
>>> math.sin(math.pi/4)
0.7071067811865475
>>>> math.degrees(math.pi)
180.0
>>>> math.factorial(9)
362880
>>> math.log(1024, 2)
10.0
```

Note : Nous apprendrons ultérieurement comment ne pas avoir à spécifier le préfixe `math.` à chaque fois.

2.6.2 Le type `complex`

Syntaxe

✎ Les complexes sont écrits en notation cartésienne formée de deux flottants. La partie imaginaire est suffixée par `j`

```
>>> 1j
1j
>>> (2+3j) + (4-7j)
(6-4j)
>>> (9+5j).real
9.0
>>> (9+5j).imag
5.0
>>> (abs(3+4j))  # module d'un complexe
5.0
```

Un module mathématique spécifique (`cmath`) leur est réservé [1] :

```
>>> import cmath
>>> cmath.phase(-1 + 0j)
3.141592653589793
>>> cmath.polar(3 + 4j)
(5.0, 0.9272952180016122)
>>> cmath.rect(1., cmath.pi/4)
(0.7071067811865476+0.7071067811865475j)
>>> cmath.sqrt(3 + 4j)
(2+1j)
```

2.7 Variables et affectation

2.7.1 Les variables

Pour stocker des données, on utilise des *variables*. Plus précisément, Python n'offre pas la notion de variable comme adresse de mémoire identifiée (comme avec le langage C) mais plutôt celle de *références d'objets* par des noms.

1. Contenant les fonctions mathématiques standard appliquées à la variable complexe.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Définition

✍ Une **variable** est un identificateur associé à une valeur. En Python, c'est une **référence d'objet**.

Tant que l'objet n'est pas modifiable (entier, flottant, chaîne, etc.), il n'y a pas de différence notable entre variable et référence. On verra que la situation change dans le cas des objets modifiables…

Une variable spéciale : `_`, utilisable uniquement en mode interactif, contient le résultat de la dernière opération [1] :

```
>>> 2 * 7 - 5
9
>>> _ + 3     # Equivalent à 9 + 3
12
```

2.7.2 L'affectation (ou assignation)

Syntaxe

✎ On *affecte* une valeur à une variable en utilisant le signe d'égalité (`=`).
Attention : l'affectation *n'a rien à voir* avec l'égalité en math !

```
a = 2
e = a == 2  # True
b = 'John Deuf'
```

Remarque

✓ Puisqu'une variable est une référence, pour ne pas confondre *affectation* et *égalité mathématique*, `b = 'John Deuf'` pourra se dire « `b` *pointe sur* 'John Deuf' ».
On illustre ce mécanisme par un cercle qui contient un nom de référence et un rectangle une valeur.

La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps par des opérations de réaffectation. Dans de telles opérations, la valeur antérieure est perdue :

```
>>> a = 3 * 7
>>> a
21
>>> b = 2 * 2
>>> b
4
>>> a = b + 5
>>> a
9
```

Le membre de droite d'une affectation étant évalué avant de réaliser l'affectation elle-même, la variable affectée peut se trouver en partie droite et c'est sa valeur avant l'affectation qui est utilisée dans le calcul :

1. Le shell interactif `ipython`, fourni entre autre par `Jupyter`, dispose d'autres variables spécifiques.

Copyright © 2017 Dunod.

```
>>> a = 2
>>> a = a + 1   # incrémentation
>>> a
3
>>> a = a - 1   # décrémentation
>>> a
2
```

Un bon exemple d'utilisation d'un même nom de variable en partie gauche et en partie droite est le calcul du terme suivant d'une suite numérique :

```
>>> u_n = 4
>>> u_n = 3 * u_n + 1
>>> u_n
13
```

2.7.3 Attention : affecter n'est pas comparer !

L'affectation a un **effet** (elle modifie l'état interne du programme en cours d'exécution) mais n'a pas de valeur (on ne peut pas l'utiliser dans une expression) :

```
>>> c = True  # c a été créé et référence la valeur True
>>> s = (c = True) and True  # On ne peut pas affecter c dans une expression
  File "<stdin>", line 1
    s = (c = True) and True
           ^
SyntaxError: invalid syntax
```

La comparaison a une **valeur** (de type `bool`) utilisable dans une expression mais n'a pas d'effet :

```
>>> c = 'a'
>>> s = (c == 'a') and True
>>> s
True
>>> c
'a'
```

2.7.4 Les variantes de l'affectation

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
>>> v = 4                # affectation simple
>>> v += 2               # affectation augmentée. Idem à v = v + 2 si v est déjà référencé
>>> v
6
>>> c = d = 8            # d reçoit 8, puis c reçoit d. Ils référencent la même donnée (alias)
>>> c, d                 # un tuple
(8, 8)
>>> e, f = 2.7, 5.1      # affectation parallèle par décapsulation d'un tuple
>>> e, f
(2.7, 5.1)
>>> a = 3
>>> a, b = a + 2, a * 2  # toutes les expressions sont évaluées avant la première affectation
>>> a, b
(5, 6)
```

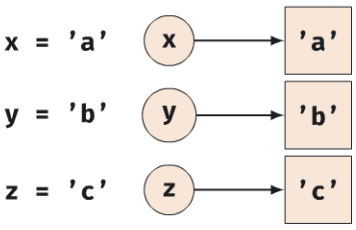
Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Remarque

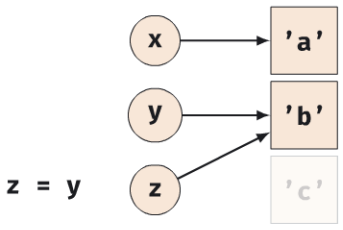
✓ Dans une affectation, le membre de gauche pointe sur le membre de droite, ce qui nécessite d'évaluer la valeur du membre de droite **avant** de l'affecter au membre de gauche. On voit dans l'affectation parallèle l'importance de cette séquence temporelle.

2.7.5 Représentation graphiques des affectations

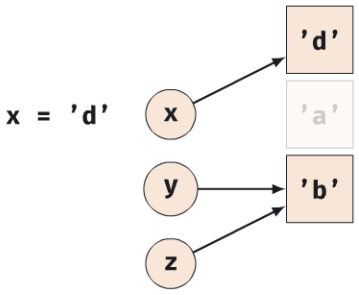
Les affectations **relient** les identificateurs aux données : si une donnée en mémoire n'est plus reliée, le ramasse-miettes (*garbage collector*) de Python la supprime automatiquement (car son nombre de références tombe à zéro [1]) :



(a) Trois affectations



(b) La donnée `'c'` est supprimée lorsque `z` référence `y`



(c) La donnée `'a'` est supprimée lorsque `x` référence `'d'`

2.7.6 Suppression d'une variable

Puisque tout est dynamique en Python, il est possible au cours de l'exécution de supprimer une variable, donc de supprimer un nom qui référence une donnée. L'instruction qui permet cela est `del`.

```
>>> a = 3
>>> del a
>>> a
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

1. CPython effectue certaines optimisations, les entiers négatifs et positifs aux alentours de zéro, couramment utilisés, sont conservés en mémoire et partagés afin de ne pas avoir à les supprimer/recréer tout le temps. D'autres implémentations de Python peuvent avoir des mécanismes de gestion de la mémoire différents.

Copyright © 2017 Dunod.

2.8 Les chaînes de caractères

2.8.1 Présentation

Définition

✍ Les **chaînes** de caractères : le type de données *non modifiable* `str` représente une séquence de caractères Unicode. *Non modifiable* signifie qu'une donnée, une fois créée en mémoire, ne pourra plus être changée ; toute transformation aboutira à la création d'une *nouvelle* valeur distincte. On utilise souvent le terme « immutable ».

Les chaînes de caractères sont des valeurs textuelles (espaces, symboles, alphanumériques...) entourées par des guillemets simples ou doubles, ou par une série de trois guillemets simples ou doubles.

L'utilisation de l'apostrophe (`'`) à la place du guillemet (`"`) autorise l'inclusion d'une notation dans l'autre. La notation entre trois guillemets permet de composer des chaînes sur plusieurs lignes contenant elles-mêmes des guillemets simples ou doubles. On verra ultérieurement que cette utilisation est très utile pour documenter des parties de programme.

Exemple :

```
>>> guillemets = "L'eau vive"
>>> apostrophes = 'Il a écrit : "Ni le mort ni le vif, mais le merveilleux !"'
>>> doc = """    forme multiligne très utile pour la documentation d'un script
    ou pour inclure un fragment de programme dans une chaîne de caractères :

for i in range(2, 2*n+1):
    if i%2 == 0:  #indice pair
        monge.insert(0, i)
    else:
        monge.append(i)"""
```

2.8.2 Les séquences d'échappement

À l'intérieur d'une chaîne, le caractère antislash (`\`) permet de donner une signification spéciale à certaines séquences de caractères.

Séquence	Signification
`\`	saut de ligne ignoré (placé en fin de ligne)
`\\`	antislash
`\'`	apostrophe
`\"`	guillemet
`\a`	sonnerie (*bip*)
`\b`	retour arrière
`\f`	saut de page
`\n`	saut de ligne
`\r`	retour en début de ligne
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\N{nom}`	caractère sous forme de code Unicode nommé

.../...

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Séquence	Signification
\uhhhh	caractère sous forme de code Unicode 16 bits sur 4 chiffres hexadécimaux
\Uhhhhhhhh	caractère sous forme de code Unicode 32 bits sur 8 chiffres hexadécimaux
\ooo	caractère sous forme de code octal sur 3 chiffres octaux
\xhh	caractère sous forme de code hexadécimal sur 2 chiffres

Séquences d'échappement des chaînes de caractères

```
>>> "\N{pound sign}  \u00A3  \U000000A3"
£  £  £
>>> "d \144 \x64"
d d d
>>> r"d \144 \x64"  # la notation r"..." (raw) désactive la signification spéciale du caractère "\"
d \144 \x64
```

Remarque

✓ On trouvera en annexe F p. 208 une liste complète des opérations et des méthodes sur les chaînes de caractères.

2.8.3 Opérations

Outre les fonctions et méthodes que nous allons voir, les quatre opérations suivantes : longueur, concaténation [1], répétition et test d'appartenance s'appliquent au type chaînes :

```
>>> len("abcde")      # Longueur (nombre de caractères dans la chaîne)
5
>>> "abc" + "defg"    # concaténation (mise bout à bout de deux chaînes ou plus)
'abcdefg'
>>> "Fi! " * 3
'Fi! Fi! Fi! '        # répétition (avec * entre une chaîne et un entier)
>>> 'y' in 'Python'   # l'opérateur 'in' teste l'appartenance d'un élément à une chaîne
True
```

2.8.4 Fonctions *vs* méthodes

On peut agir sur une chaîne en utilisant des *fonctions* (notion procédurale) communes à tous les types séquences ou conteneurs, ou bien des *méthodes* (notion objet) spécifiques aux chaînes :

```
>>> len('Les auteurs')     # syntaxe d'une fonction
11
>>> "abracadabra".upper()  # syntaxe d'une méthode (notation pointée)
"ABRACADABRA"
```

2.8.5 Méthodes de test de l'état d'une chaîne

Il s'agit de méthodes à valeur booléenne, c'est-à-dire qu'elles retournent la valeur `True` ou `False`.

Syntaxe

✎ La notation entre crochets *[xxx]* indique un élément optionnel, que l'on peut donc omettre lors de l'utilisation de la méthode.

1. C'est-à-dire mise bout à bout de deux chaînes (ou plus) .

Copyright © 2017 Dunod.

Voici quelques exemples de ces méthodes de test. Une liste complète se trouve en annexe (cf. annexe F p. 208).

```
>>> 'Le petit PRINCE'.isupper()           # tout est en majuscule
False
>>> 'Le Petit Prince'.istitle()           # chaque mot commence par une majuscule
True
>>> 'Prince'.isalpha()                    # ne contient que des caractères alphabétiques
True
>>> 'Un'.isdigit()                        # ne contient que des caractères numériques
False
>>> 'Le Petit Prince'.startswith('Le ')   # commence par...
True
>>> 'Le Petit Prince'.endswith('prince')  # ... finit par
False
```

2.8.6 Méthodes retournant une nouvelle chaîne

Comme les chaînes sont immutables (c'est-à-dire que leur contenu ne peut pas changer), les méthodes qui les modifient retournent de *nouvelles* chaînes. En voici quelques exemples :

```
>>> 'Le petit PRINCE'.lower()           # tout en minuscule
'le petit prince'
>>> 'Le petit PRINCE'.upper()           # tout en majuscule
'LE PETIT PRINCE'
>>> 'Le petit PRINCE'.swapcase()        # inverser la casse
'lE PETIT prince'
>>> 'Le petit PRINCE'.center(31, '~')   # chaîne centrée
'~~~~~~~~Le petit PRINCE~~~~~~~~'
>>> 'Le petit PRINCE'.rjust(31, '^')    # chaîne justifiée à droite
'^^^^^^^^^^^^^^^^Le petit PRINCE'
>>> '  Le petit Prince '.lstrip('e L')  # suppression de caractères en début de chaîne
'petit Prince '
>>> 'Le petit Prince'.split()           # découpe la chaîne suivant le séparateur
                                        # (séquence d'espaces par défaut)
['Le', 'petit', 'Prince']
```

2.8.7 Méthode retournant un indice

`find(sub[, start[, stop]])` : renvoie l'index de la chaîne `sub` dans la sous-chaîne `start` à `stop`, sinon renvoie `-1`. `rfind()` effectue le même travail en commençant par la fin. `index()` et `rindex()` font de même mais produisent une erreur (*exception*) si la chaîne `sub` n'est pas trouvée :

```
>>> 'Le petit Prince'.find('Pr')
9
>>> 'Le petit Prince'.index('bad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

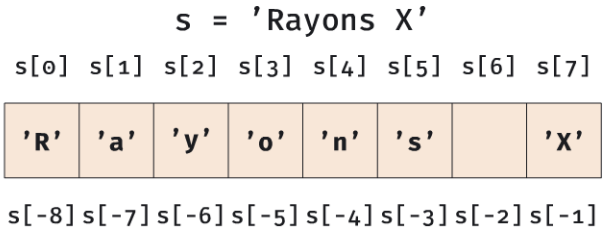
2.8.8 Indexation simple

Syntaxe

✎ L'opérateur d'indexation utilise la notation `[index]` dans lequel `index` est un entier signé *qui commence à* `0` et indique la position d'un caractère.

L'utilisation de valeurs d'index négatives permet d'accéder aux caractères par la fin de la chaîne :

```
>>> s = "Rayons X"   # len(s) ==> 8
>>> s[0]             # premier caractère
'R'
>>> s[2]             # troisième caractère
'y'
>>> s[-1]            # dernier caractère
'X'
>>> s[-3]            # antepénultième caractère
's'
```



2.8.9 Extraction de tranches

Syntaxe

✎ L'opérateur d'extraction de tranche `[début:fin]` ou `[début:fin:pas]`, dans lequel `début` et `fin` sont des index de caractères et `pas` est un entier signé, permet d'extraire des tranches [1].

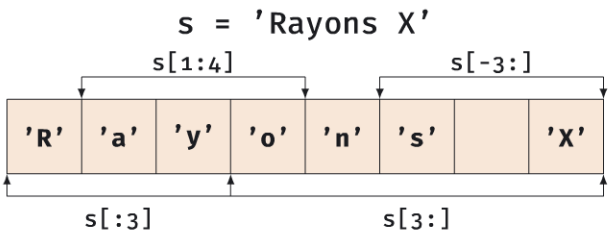
L'omission de `début` ou de `fin` permet de spécifier du début (ou respectivement de la fin) de la chaîne.

Par exemple :

```
>>> s = "Rayons X"   # len(s) ==> 8
>>> s[1:4]           # de l'index 1 compris à 4 non compris
'ayo'
>>> s[-3:]           # de l'index -3 compris à la fin
's X'
>>> s[:3]            # du début à l'index 3 non compris
'Ray'
>>> s[3:]            # de l'index 3 compris à la fin
'ons X'
>>> s[::2]           # du début à la fin, de 2 en 2
'Ryn '
>>> s[::-1]          # du début à la fin en pas inverse (retournement)
'X snoyaR'
```

1. On utilise habituellement le mot « tranche » au sens de « sous-chaîne de caractères ».

Copyright © 2017 Dunod.



2.8.10 Opérateur de formatage des chaînes

Pour mettre en forme une chaîne de caractères (par exemple en vue de l'afficher), plusieurs solutions existent. Voici la syntaxe la plus simple [1] qui utilise l'opérateur `%`.

Syntaxe

```
format % valeurs
```

Où `format` est une chaîne contenant des spécificateurs de format et `valeurs` est un tuple ou un dictionnaire contenant une ou plusieurs valeurs. Chaque spécificateur est formé du caractère `%` suivi d'un caractère dont la signification est donnée dans le tableau ci-dessous. À l'affichage, cette spécification est remplacée, dans l'ordre, par son correspondant dans `valeurs`.

Par exemple :

```
>>> ch = 'trois'
>>> "%s %d %s" % ('One', 2, ch+' Go!')
'One 2 trois Go!'
```

Les spécificateurs de format `%` sont donnés dans le tableau suivant :

Caractère	Format de sortie
`d, i`	entier décimal signé
`u`	entier décimal non signé
`o`	entier octal non signé (sans le préfixe `0o`)
`x, X`	entier hexadécimal non signé écrit en minuscule (x) ou en majuscule (X)
`e, E`	flottant forme exponentielle
`f, F`	flottant forme décimale
`g, G`	comme `e` si l'exposant est supérieur à 4, comme `f` sinon
`c`	caractère simple
`s`	chaîne interprétée par l'application de la fonction `str()`
`r`	chaîne interprétée par l'application de la fonction `repr()`
`%`	le caractère littéral `%`

Par exemple :

```
>>> "%s" % (1 / 3.0)
'0.3333333333333333'
>>> "%.2f" % (1 / 3.0)
'0.33'
>>> "%(toto)s" % dict(toto=12)   # formatage à l'aide d'un dictionnaire
'12'
```

1. Héritée du langage C.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2.8.11 Affichage formaté

La méthode `format()` permet de contrôler finement la création de chaînes formatées. On l'utilisera pour un affichage *via* `print()`, pour un enregistrement dans un fichier *via* `f.write()`, ou dans d'autres cas de flux de sortie. Chaque paire d'accolades désigne un champ qui est rempli avec la valeur d'un des paramètres de la méthode `format()`, suivant les directives données entre les accolades.

Remplacements simples :

```
print("{} {} {}".format("zéro", "un", "deux"))     # zéro un deux

# formatage d'une chaîne pour usages ultérieurs
chain = "{2} {0} {1}".format("zéro", "un", "deux")

print(chain)        # affichage : deux zéro un

print("Je m'appelle {}".format("Bob"))      # Je m'appelle Bob
print("Je m'appelle {{{}}}".format("Bob"))  # Je m'appelle {Bob}
print("{}".format("-"*10))                  # ----------

# adaptation du message de saisie :
msg = "Saisissez la {}e valeur : "
lst = []
for i in range(1, 10):
    lst.append(input(msg.format(i)))
```

Remplacements avec champs nommés :

```
a, b = 5, 3
print("The story of {c} and {d}".format(c=a+b, d=a-b))  # The story of 8 and 2
```

Formatages à l'aide d'une liste [cf. p. 43]. Le premier nombre indice l'argument de `format` à utiliser, le nombre entre crochets spécifie l'indice dans cet argument :

```
stock = ['papier', 'enveloppe', 'chemise', 'encre', 'buvard']
print("Nous avons de l'{0[3]} et du {0[0]} en stock\n".format(stock))
# Nous avons de l'encre et du papier en stock
```

Formatages à l'aide d'un dictionnaire [cf. p. 49]. Le premier nombre indice l'argument de `format()` à utiliser, le texte entre crochets spécifie la clé dans cet argument :

```
print("My name is {0[name]}".format(dict(name='Fred')))  # My name is Fred

d = dict(poids = 12000, animal = 'éléphant')
print("L'{0[animal]} pèse {0[poids]} kg\n".format(d))  # L'éléphant pèse 12000 kg
```

Remplacement avec attributs nommés [cf. p. 96]. On utilise alors la notation pointée des espaces de noms, appliquée aux arguments de `format()` :

```
import math
import sys

print("math.pi = {.pi}, epsilon = {.float_info.epsilon}".format(math, sys))
# math.pi = 3.14159265359, epsilon = 2.22044604925e-16
```

Copyright © 2017 Dunod.

Conversions textuelles, `str()` et `repr()`. La fonction `str()` produit une représentation orientée utilisateur alors que `repr()` produit une représentation littérale :

```
>>> print("{0!s}   {0!r}".format("texte\n"))
texte
   'texte\n'
```

Formatages numériques, nombres entiers dans différentes bases, nombres flottants sous différentes notations et différentes précisions :

```
s = "int :{0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
print(s)  # int :42; hex: 2a; oct: 52; bin: 101010
s = "int :{0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
print(s)  # int :42; hex: 0x2a; oct: 0o52; bin: 0b101010

n = 100
pi = 3.1415926535897931
k = -54

print("{:.4e}".format(pi))         # 3.1416e+00
print("{:g}".format(pi))           # 3.14159
print("{:.2%}".format(n/(47*pi)))  # 67.73% - Ce format évite de multiplier le ratio par 100

msg = "Résultat sur {:d} échantillons : {:.2f}".format(n, pi)
print(msg)  # Résultat sur 100 échantillons : 3.14

msg = "{0.real} et {0.imag} sont les composantes du complexe {0}".format(3-5j)
print(msg)  # 3.0 et -5.0 sont les composantes du complexe (3-5j)

print("{:+d} {:+d}".format(n, k))  # +100 -54 (on force l'affichage du signe)

print("{:,}".format(1234567890.123))  # 1,234,567,890.12
```

Formatages divers :

```
>>> s = "The sword of truth"
>>> print("[{}]".format(s))
[The sword of truth]
>>> print("[{:25}]".format(s))
[The sword of truth       ]
>>> print("[{:>25}]".format(s))
[       The sword of truth]
>>> print("[{:^25}]".format(s))
[   The sword of truth    ]
>>> print("[{:-^25}]".format(s))
[---The sword of truth----]
>>> print("[{:.<25}]".format(s))
[The sword of truth.......]
>>> lng = 12
>>> print("[{}]".format(s[:lng]))
[The sword of]
>>> m = 123456789
>>> print("{:0=12}".format(m))
000123456789
```

On pourra se reporter à l'encadré « Formatage » du mémento, en page intérieure de la couverture.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2.9 Les types binaires

Python 3 propose deux types de données binaires : `bytes` (immutable) et `bytearray` (mutable).

```
>>> type('Animal')
str
>>> type(b'Animal')
bytes
>>> type(bytearray(b'Animal'))
bytearray
```

Une donnée binaire contient une suite, éventuellement vide, d'octets, c'est-à-dire une suite d'entiers non signés sur 8 bits (compris chacun dans l'intervalle [0…255]). Ces types « *à la C* » sont bien adaptés pour stocker de grandes quantités de données ou encore des données ayant une structure définie précisément au niveau des octets ou des bits. De plus, Python fournit des moyens de manipulation efficaces de ces types [1].

Les deux types sont assez semblables au type `str` et possèdent la plupart de ses méthodes. Le type modifiable `bytearray` possède aussi des méthodes communes au type `list` que nous verrons bientôt (cf. § 4.2 p. 43).

2.10 Les entrées-sorties

L'utilisateur a besoin d'interagir avec le programme. En mode « console » (nous aborderons les interfaces graphiques ultérieurement), on doit pouvoir *saisir* ou *entrer* des informations, ce qui est généralement fait depuis une **lecture** au clavier. Inversement, on doit pouvoir *afficher* ou *sortir* des informations, ce qui correspond généralement à une **écriture** sur l'écran.

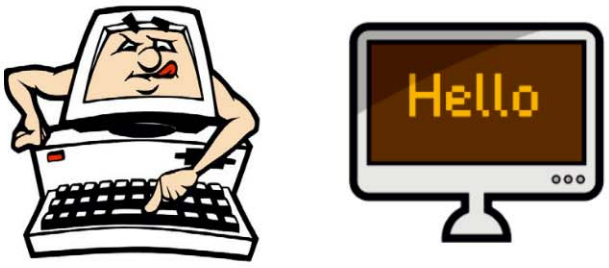


FIGURE 2.1 – Les entrées (principalement au clavier) et les sorties (principalement à l'écran)

2.10.1 Les entrées

Il s'agit de réaliser une *saisie* au clavier : la fonction standard `input()` interrompt le programme, affiche une éventuelle invite à l'écran et attend que l'utilisateur entre une donnée au clavier (affichée à l'écran) et la valide par Entrée.

La fonction `input()` effectue toujours une saisie en *mode texte* (la valeur retournée est une chaîne) dont on peut ensuite changer le type (on dit aussi « transtyper ») :

1. En l'occurrence le module standard `struct`.

Copyright © 2017 Dunod.

```
>>> f = input("Entrez un flottant : ")
Entrez un flottant : 12.345
>>> type(f)
str
>>> g = float(f)   # transtypage
>>> type(g)
float
```

Une fois la donnée numérique convertie dans son type « naturel » (`float`), on peut l'utiliser pour faire des calculs.

On rappelle que Python est un langage *dynamique* (les variables peuvent changer de type au gré des affectations) mais néanmoins *fortement typé* (contrôle de la cohérence des types) :

```
>>> i = input("Entrez un entier : ")
Entrez un entier : 3
>>> i
'3'
>>> iplus = int(input("Entrez un entier : ")) + 1
Entrez un entier : 3
>>> iplus
4
>>> ibug = input("Entrez un entier : ") + 1
Entrez un entier : 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

On voit sur l'exemple précédent que Python n'autorise pas d'additionner une variable de type entier avec une variable de type chaîne.

2.10.2 Les sorties

En mode « calculatrice », Python *lit-évalue-affiche* (☞ Fig. 2.1, p. 15), mais la fonction `print()` reste indispensable aux affichages dans les scripts. Elle se charge d'afficher la représentation textuelle des informations qui lui sont données en paramètre, en plaçant un blanc séparateur entre deux informations, et en faisant un retour à la ligne à la fin de l'affichage (le séparateur et la fin de ligne peuvent être modifiés) :

```
>>> print('Hello World!')
Hello World!
>>> print()            # affiche une ligne blanche

>>> a, b = 2, 5
>>> print('Somme :', a + b, ';', a - b, 'est la différence et', a * b, 'le produit.')
Somme : 7 ; -3 est la différence et 10 le produit.
>>>  :
>>> print(a, end='@')  # affiche un autre caractère qu'un espace en fin de ligne
2@
```

La fonction `print()` produit des affichages de chaînes et de variables, tant sur les consoles de sortie que dans des fichiers. Très fréquemment, nous aurons besoin d'affichages *formatés*. Nous avons déjà vu une méthode simple avec l'opérateur `%` [cf. p. 29] et une syntaxe plus détaillée [cf. p. 30]).

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2.11 Exercices

1. On fournit une variable numérique flottante `a` avec une valeur quelconque. Écrire l'expression logique qui est vraie lorsque `a` est dans l'intervalle fermé $[10,20]$.

2. Écrire un programme qui, à partir de la saisie d'un rayon et d'une hauteur, calcule le volume d'un cône droit (on rappelle que le volume du cône est donné par : $V = \frac{\pi r^2 h}{3}$ où r et h sont le rayon et la hauteur du cylindre).

3. Soit une variable `nbessais` contenant un nombre de tentatives déjà effectuées de saisie d'une valeur, ne pouvant dépasser 5 essais. Soit une variable `v` contenant un nombre entier que l'on veut strictement positif, divisible par 3 (le reste de sa division entière par 3 doit être nul) et strictement inférieur à 100.
   Donner l'expression logique qui est vraie lorsque la valeur n'est pas valide et qu'il est encore possible de tenter une nouvelle saisie.

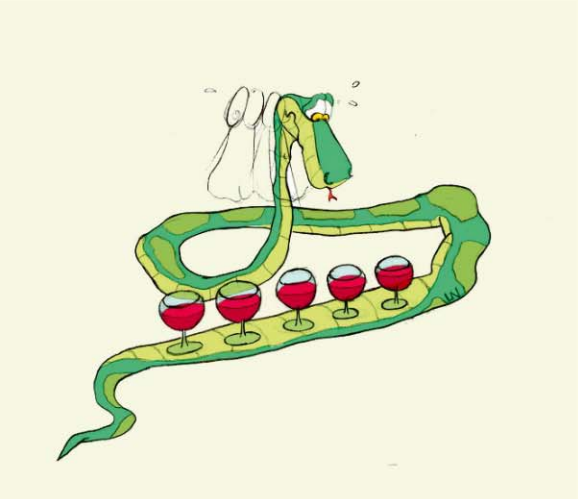
4. À partir d'une variable `s` contenant `"Dark side of the moon"`, écrire l'expression permettant, à partir de cette variable, de construire cette même chaîne avec la première lettre de chaque mot en majuscules, encadré de caractères `=`, l'ensemble sur une largeur de 60 caractères :

```
'==================Dark Side Of The Moon==================='
```

Copyright © 2017 Dunod.

Chapitre 3

Contrôle du flux d'instructions



Un script Python est formé d'une suite d'instructions exécutées en séquence de haut en bas, c'est le flux normal d'instructions.

Chaque ligne d'instructions est formée d'une ou de plusieurs lignes physiques. L'étalement d'une instruction sur plusieurs lignes peut être implicite, en raison d'expressions ouvertes avec des caractères ouvrants `[` ou `(` ou `{` qui n'ont pas encore été fermées, ou bien explicite, par l'écriture d'un antislash `\` en fin de ligne pour signifier sa continuation. Sur une ligne physique, il est possible de placer plusieurs instructions séparées par des point-virgules `;`, toutefois ceci est rarement utilisé car peu lisible.

Ce chapitre explique comment ce flux peut être modifié pour *choisir* ou *répéter* des portions de code en utilisant des « instructions composées ».

3.1 Instructions composées

Pour identifier les instructions composées, Python utilise la notion d'*indentation significative*. Cette syntaxe, légère et visuelle, met en lumière un bloc d'instructions et permet d'améliorer grandement la présentation et donc la lisibilité des programmes sources.

Syntaxe

Une instruction composée se compose :

- d'une **ligne d'introduction** terminée par le caractère « deux-points » (`:`) ;
- d'un **bloc d'instructions indenté par rapport à la ligne d'introduction**. On utilise par convention quatre espaces par indentation et on n'utilise pas les tabulations mais uniquement les espaces.

Copyright © 2017 Dunod.

Exemple d'instruction composée simple :

```
ph = float(input("PH ? "))
if ph < 7:
    print("Le potentiel hydrogène (pH) est inférieur à 7.")
    print("C'est un acide.")
if ph > 7:
    print("Le potentiel hydrogène (pH) est supérieur à 7.")
    print("C'est une base.")
if ph == 7:
    print("Le potentiel hydrogène (pH) est exactement 7.")
    print("La solution est neutre.")
```

Exemple d'instruction composée imbriquée :

```
t = float(input("Température (°C) ? "))
print("Température 't' en degrés Celsius")
if t <= 0:
    print('Négative ou nulle : risque de gel')
else:
    print('Positive')
    if t > 25:
        print('Plus de 25 °C')
        print('Prévoir tee-shirt ou veste légère')
    elif t > 18 :
        print('Douce mais sans plus')
    else
        print('Mais sortez couverts')
print("Évaluation terminée")
```

Attention

☢ Toutes les instructions au même niveau d'indentation appartiennent au même bloc d'instructions, jusqu'à ce que l'indentation redevienne inférieure à ce niveau.

3.2 Choisir

Afin d'aiguiller différemment le flux normal des instructions, on utilise des *instructions conditionnelles*, qui permettent de contrôler des alternatives d'exécution entre différents blocs d'instructions.

3.2.1 Choisir : `if - [elif] - [else]`

Exemple de contrôle d'une alternative :

```
>>> x = 5
>>> if x < 0:
...     print("x est négatif")
... elif x % 2 != 0:
...     print("x est positif et impair")
...     print ("ce qui est bien aussi !")
... else:
...     print("x n'est pas négatif et est pair")
...

x est positif et impair
ce qui est bien aussi !
```

Copyright © 2017 Dunod.

Il est possible d'enchaîner plusieurs blocs `elif` mais il n'y a qu'un seul bloc `else`.

Test d'une valeur booléenne :

```
>>> # Attention de bien choisir le nom de la variable booléenne !
>>> x = 8
>>> estPair = (x % 2 == 0)
>>> estPair
True
>>> if estPair:  # mieux que "if estPair == True:"
...     print('La condition est vraie')
...
La condition est vraie
```

3.2.2 Syntaxe compacte d'une alternative

Pour trouver, par exemple, le minimum de deux nombres, on peut utiliser l'opérateur *ternaire* :

```
>>> x = 4
>>> y = 3
>>> if x < y:                          # écriture classique
...     plus_petit = x
... else:
...     plus_petit = y
...
>>> print("Plus petit : ", plus_petit)
Plus petit :  3
>>> plus_petit = x if x < y else y       # utilisation de l'opérateur ternaire
>>> print("Plus petit : ", plus_petit)
Plus petit :  3
```

Remarque

✓ L'opérateur ternaire est une *expression* qui fournit une valeur que l'on peut utiliser dans une affectation ou un calcul. Seule l'expression du résultat utilisé est évaluée.

3.3 Boucles

Notions de conteneur et d'itérable

Définition

✍ Un **conteneur** désigne un type de données permettant de stocker un ensemble d'autres données, en ayant ou non, suivant le type du conteneur, une notion d'ordre entre ces données.

Définition

✍ Un **itérable** désigne un type de conteneur que l'on peut parcourir élément par élément ou qui est capable de fournir des séquences de valeurs à la demande.

Pour parcourir ces conteneurs, nous nous servirons parfois de l'instruction `range()`, qui fournit un moyen commode pour générer une série de valeurs entières.

Par exemple :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
>>> uneListe = list(range(6))
>>> uneListe
[0, 1, 2, 3, 4, 5]
```

Ces notions seront étudiées plus en détail au chapitre 4, p. 43.

Python propose deux sortes de boucles.

3.3.1 Répéter : `while`

Répéter une portion de code (chaque répétition est appelée une *itération*) tant qu'une expression booléenne est vraie :

```
>>> x, cpt = 257, 0
>>> sauve = x
>>> while x > 1:
...     x = x // 2     # division avec troncature
...     cpt = cpt + 1  # incrémentation
...
>>> print("Approximation de log2 de", sauve, ":", cpt)
Approximation de log2 de 257 : 8
```

Utilisation classique : la *saisie filtrée* d'une valeur numérique (on doit *préciser le type* de la saisie, car on se rappelle que `input()` retourne une *chaîne* correspondant à la saisie de l'utilisateur) :

```
n = int(input('Entrez un entier [1 .. 10] : '))
while not(1 <= n <= 10):
    n = int(input('Entrez un entier [1 .. 10], S.V.P. : '))
```

Si la saisie est compliquée, on peut utiliser une variable « drapeau » (c'est-à-dire booléenne) :

```
saisie_ok = False
while not saisie_ok :    # tant que saisie_ok est False, faire :
    a = int(input("Entrez un nombre entier de 1 à 100 divisible par 3 : "))
    b = int(input("Entrez un nombre entier pair supérieur à " + str(a) + ": "))
    saisie_ok = (1 <= a <=100) and (b % 2 == 0) and (a < b)
```

3.3.2 Parcourir : `for`

Parcourir un itérable permet d'accéder tour à tour à chaque valeur afin de la traiter dans le corps de la boucle :

```
>>> for lettre in "ciao":   # On peut itérer sur les caractères des chaînes
...     print(lettre, lettre.upper())
...
c C
i I
a A
o O
>>> for x in [2, 'a', 3.14]: # Notation pour une liste de valeurs (cf. chap. 4)
...     print(x)              # Le typage dynamique de Python permet à x de recevoir à
...     print(2 * x)          # chaque itération une valeur d'un type différent
...
2
```

Copyright © 2017 Dunod.

```
4
a
aa
3.14
6.28
>>> for i in range(6):       # Générateur de séquences d'entiers à la demande
...     print(i, i ** 2)
...
0 0
1 1
2 4
3 9
4 16
5 25
>>> nb_voyelles = 0
>>> for lettre in "Python est un langage fort sympa":
...     if lettre.lower() in "aeiouy":
...         nb_voyelles = nb_voyelles + 1
...
>>> nb_voyelles
10
```

3.4 Ruptures de séquences

3.4.1 Interrompre une boucle : `break`

Sort immédiatement du corps de la boucle `for` ou `while` en cours contenant le `break` et passe à l'instruction suivante après la boucle :

```
for x in range(1, 11):
    if x == 5:
        break
    print(x, end=" ") # end=" " remplace le retour à la ligne du print() par un simple espace

print("Boucle interrompue pour x =", x)
```

Ce qui produit :

```
1 2 3 4 Boucle interrompue pour x = 5
```

Interruption dans les boucles

Signalons une syntaxe des boucles spécifique à Python. Les boucles `while` et `for` peuvent posséder une clause `else` qui ne s'exécute que si la boucle se termine « normalement », c'est-à-dire sans interruption par `break`.

3.4.2 Court-circuiter une boucle : `continue`

Passe immédiatement à l'itération suivante de la boucle `for` ou `while` en cours contenant l'instruction ; reprend à la ligne de l'en-tête de la boucle pour préparer l'itération suivante. Ceci permet d'ignorer des valeurs ou des cas, selon des critères choisis, lors de traitements répétitifs :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
>>> for x in range(1, 11):
...     if x == 5:
...         continue
...     print(x, end=" ")
...
1 2 3 4 6 7 8 9 10
>>> # la boucle a sauté la valeur 5
```

3.4.3 Traitement des erreurs : les exceptions

Afin de rendre les applications plus robustes, il est nécessaire de gérer les erreurs d'exécution des parties sensibles du code.

Lorsqu'une erreur se produit, elle traverse toutes les couches de code comme une bulle d'air remonte à la surface de l'eau. Quand elle atteint la surface sans être interceptée par le mécanisme des *exceptions*, le programme s'arrête et l'erreur est affichée. Le *traceback* (message complet d'erreur affiché) précise l'ensemble des couches traversées. On décrira plus précisément (cf. annexe E p. 190) comment ces traces peuvent être utilisées pour trouver les origines des erreurs dans les programmes.

Définition

✍ **Gérer une exception** permet d'intercepter une erreur pour éviter un arrêt du programme.

Une exception sépare d'un côté la séquence d'instructions à exécuter lorsque tout se passe bien et, d'un autre côté, une ou plusieurs séquences d'instructions à exécuter en cas d'erreur.

Lorsqu'une erreur survient, un *objet exception* est passé au mécanisme de propagation des exceptions, et l'exécution est transférée à la séquence de traitement *ad hoc*.

Le mécanisme s'effectue donc en deux phases :
- la **levée d'exception** lors de la détection d'erreur ;
- le **traitement** approprié.

Syntaxe

✎ La séquence normale d'instructions est placée dans un bloc `try`.
Si une erreur est détectée (levée d'une exception), elle est traitée dans le bloc `except` approprié.

```
from math import sin

for x in range(-4, 5):  # -4, -3, -2, -1, 0, 1, 2, 3, 4
    try:
        print('{:.3f}'.format(sin(x)/x), end=" ")
    except ZeroDivisionError:   # toujours fournir un type d'exception
        print(1.0, end=" ")     # gère l'exception en 0
# -0.189 0.047 0.455 0.841 1.0 0.841 0.455 0.047 -0.189
```

Les exceptions levées par Python sont organisées en une arborescence de classes (familles) ayant pour ancêtre commun une classe `Exception`, arborescence décrite en annexe (cf. annexe E p. 201).

Syntaxe complète d'une exception :

```
try:
    ...                 # séquence normale d'exécution
except <exception_1> as e1:
```

Copyright © 2017 Dunod.

```
    ...                 # traitement de l'exception 1
except <exception_2> as e2:
    ...                 # traitement de l'exception 2
...
else:
    ...                 # clause exécutée en l'absence d'erreur
finally:
    ...                 # clause finale toujours exécutée, avec ou sans erreur
```

L'instruction **raise** permet à l'utilisateur de lever *volontairement* une exception [1]. On peut trouver l'instruction à tout endroit du code, pas seulement dans un bloc **try** :

```
x = 2
if not(0 <= x <= 1):
    raise ValueError("x n'est pas dans [0 .. 1]")
```

3.5 Exercices

1. Dans une boucle **while**, entrer des prix d'achat HT en euros de produits (saisir 0 pour terminer). Afficher, pour chaque prix, la valeur TTC correspondante pour un taux de TVA de 20 %, calculer la somme TTC en cours et compter combien de produits valaient au moins 100 €. Après la dernière saisie, afficher la somme finale TTC et le nombre d'achats supérieurs à 100 € TTC.

2. L'utilisateur saisit un entier positif et le programme annonce combien de fois de suite cet entier est divisible par 2.

3. ✠ L'utilisateur saisit un entier supérieur à 1 et le programme affiche, s'il y en a, tous ses diviseurs propres *sans répétition* ainsi que leur nombre. S'il n'y en a pas, il indique qu'il est premier.
   On rappelle qu'un *diviseur propre* de n est un diviseur quelconque de n, n exclu.

   Par exemple

   ```
   Entrez un entier strictement positif : 12
   Diviseurs propres sans répétition de 12 : 2 3 4 6 (soit 4 diviseurs propres)

   Entrez un entier strictement positif : 13
   Diviseurs propres sans répétition de 13 : aucun ! C'est un nombre premier
   ```

4. ✠ Saisir un entier entre 1 et 3 999 (pourquoi cette limitation ?). L'afficher sous forme de nombre romain. Rappel sur la numérotation romaine, qui dispose de 7 symboles :

Symbole	Valeur
I	1
V	5
X	10
L	50
C	100
D	500
M	1 000

1. Utilisé dans un bloc **except** sans spécifier d'argument, **raise** permet de ne pas bloquer une exception dans ce bloc et de la redéclencher pour la propager à un niveau supérieur après que le traitement local de l'erreur a été effectué.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Le tableau suivant donne les règles de composition d'un nombre romain.

Règle	Exemple
Règle additive pour atteindre un nombre	`123 => C XX III`
Procéder puissance de dix par puissance de dix	`3278 => MMM CC LXX VIII`
Ne pas utiliser plus de trois symboles identiques. Utiliser la règle soustractive	`400 => CD au lieu de CCCC`
Pour la règle soustractive, n'utiliser que le symbole immédiatement avant	`1490 => MXD`

Copyright © 2017 Dunod.

Chapitre 4

Conteneurs standard

Le chapitre 2 a présenté les types de données simples, mais Python offre beaucoup plus : les conteneurs.

De façon générale, un conteneur est un objet composite destiné à contenir d'autres objets. Ce chapitre détaille les séquences, les tableaux associatifs et les ensembles.

4.1 Séquences

Définition

✍ Une **séquence** est un conteneur *ordonné* d'éléments *indexés par des entiers* indiquant leur position dans le conteneur.

Python dispose de trois types prédéfinis de séquences utilisés couramment :

- les **chaînes** (vues précédemment) ;
- les **listes** ;
- les **tuples** [1].

4.2 Listes

Remarque

✓ On trouvera en annexe F p. 209 une liste complète des opérations et des méthodes sur les listes.

1. Le mot « tuple » n'est pas vraiment un anglicisme mais plutôt un néologisme informatique. Nous l'utiliserons de préférence à *n-uplet*.

Copyright © 2017 Dunod.

4.2.1 Définition, syntaxe et exemples

Définition

Une **liste** est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Syntaxe

Éléments séparés par des virgules, et entourés de crochets.

Exemples simples de listes :

```
couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
print(couleurs)         # ['trèfle', 'carreau', 'coeur', 'pique']
couleurs[1] = 14
print(couleurs)         # ['trèfle', 14, 'coeur', 'pique']
list1 = ['a', 'b']
list2 = [4, 2.718]
list3 = [list1, list2]  # liste de listes
print(list3)            # [['a', 'b'], [4, 2.718]]
```

4.2.2 Initialisations et tests d'appartenance

Construction d'une liste vide et d'une liste répétant n fois une séquence de base :

```
>>> truc = []
>>> truc
[]
>>> machin = [0.0] * 3
>>> machin
[0.0, 0.0, 0.0]
```

Utilisation de l'itérateur d'entiers `range()` :

```
>>> liste_1 = list(range(4))   # range() : générateur de séquences d'entiers à la demande
>>> liste_1
[0, 1, 2, 3]
>>> liste_2 = list(range(4, 8))
>>> liste_2
[4, 5, 6, 7]
>>> liste_3 = list(range(2, 9, 2))
>>> liste_3
[2, 4, 6, 8]
```

Utilisation de l'opérateur d'appartenance (`in`) :

```
>>> 2 in liste_1, 8 in liste_2, 6 not in liste_3
(True, False, False)
```

4.2.3 Méthodes modificatrices

Voici quelques exemples de méthodes de modification des listes. Une liste complète se trouve en annexe (cf. annexe F p. 209).

Copyright © 2017 Dunod.

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()             # tri de la liste sur place
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)         # ajout d'un élément à la fin
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> nombres.reverse()          # inversion des éléments de la liste
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.remove(38)         # suppression d'une valeur (ou bien : del nombres[2])
>>> nombres
[12, 72, 25, 17, 10]
>>> nombres.extend([1, 2, 3])  # ajout d'une séquence d'éléments à la fin
>>> nombres
[12, 72, 25, 17, 10, 1, 2, 3]
```

4.2.4 Manipulation des index et des « tranches »

Syntaxe

✎ La manipulation des index et des tranches utilise la même syntaxe que celle déjà vue pour les chaînes [cf. p. 28].

Si on veut supprimer, remplacer ou insérer *plusieurs* éléments dans une liste, on peut indiquer une tranche dans le membre de gauche d'une affectation et fournir une séquence dans le membre de droite.

```
>>> mots = ['jambon', 'sel', 'miel', 'confiture', 'beurre']
>>> mots[2:4] = []        # effacement par affectation d'une liste vide
>>> mots
['jambon', 'sel', 'beurre']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
>>> mots[2:2] = ['miel']  # insertion en 3è position
>>> mots
['jambon', 'mayonnaise', 'miel', 'poulet', 'tomate']
```

4.3 Tuples

Définition

✍ Un **tuple** est une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes.

Syntaxe

✎ Les éléments d'un tuple sont séparés par des virgules, et entourés de parenthèses.

Un tuple ne comportant qu'un seul élément (un *singleton*) doit être obligatoirement noté avec une virgule terminale.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
>>> mon_tuple = ('a', 2, [1, 3])       # tuple de trois éléments
>>> ton_tuple = 'un', 'deux', 'trois'  # tuple de trois éléments
>>> s = (2.718,)                       # singleton
>>> t = 'toto',                        # singleton
>>> v = (,)                            # tuple vide
>>> w = tuple()                        # tuple vide
```

— L'indexage des tuples s'utilise comme celui des listes et des chaînes [cf. p. 28].
— Le parcours des tuples est plus rapide que celui des listes.
— Ils sont utiles pour définir des constantes.

Attention

Comme les chaînes de caractères, une fois construits, les tuples ne sont pas modifiables !

```
>>> mon_tuple = (1, 2)
>>> mon_tuple[1] = 3         # attention ne pas modifier un tuple !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Les opérations des objets de type séquentiel

Les types prédéfinis de séquences Python (chaîne, liste et tuple) ont en commun les opérations résumées dans le tableau suivant, où `s` et `t` désignent deux séquences du même type et `i`, `j` et `k` des entiers :

l'opération	son effet
`x in s`	`True` si `s` contient `x`, `False` sinon
`x not in s`	`True` si `s` ne contient pas `x`, `False` sinon
`s + t`	concaténation de `s` et `t`
`s * n`, `n * s`	`n` copies (superficielles) concaténées de `s`
`s[i]`	i^e élément de `s` (à partir de 0)
`s[i:j]`	tranche de `s` de `i` (inclus) à `j` (exclu)
`s[i:j:k]`	tranche de `s` de `i` à `j` avec un pas de `k`
`len(s)`	nombre d'éléments de `s`
`max(s)`, `min(s)`	plus grand, plus petit élément de `s`
`s.index(i)`	indice de la 1^{re} occurrence de `i` dans `s`
`s.count(i)`	nombre d'occurrences de `i` dans `s`
`sum(s)`	somme des éléments de `s`

4.4 Séquences de séquences

Les séquences, comme du reste les autres conteneurs, peuvent être imbriquées. Par exemple :

```
>>> un_tuple = (1, 2, 3)
>>> sequences = [un_tuple, [4, 5]]
>>> for sequence in sequences:
...     for item in sequence:
...         print(item, end=' ')
```

Copyright © 2017 Dunod.

```
...     print()
...
1 2 3
4 5
```

4.5 Retour sur les références

Nous avons déjà vu l'opération d'affectation, apparemment innocente. Or celle-ci peut être source de complications en raison du partage de valeurs par plusieurs variables.

```
i = [1, 2, 3]
msg = "Quoi de neuf ?"
e = 2.718
```

Dans l'exemple ci-dessus, les affectations réalisent plusieurs opérations :

— création en mémoire d'un objet du type approprié (membre de droite) ;
— stockage de la donnée dans l'objet créé ;
— création d'un nom de variable (membre de gauche) ;
— association de ce nom de variable avec l'objet contenant la valeur.

Copie « simple »

Une conséquence de ce mécanisme est que, si un objet modifiable (mutable) est affecté à plusieurs variables, tout changement de l'objet *via* une variable sera visible sur tous les autres. Comme nous le verrons de façon plus détaillée [cf. p. 120], Python possède des outils d'autoanalyse, en particulier la fonction `id()` qui fournit l'identifiant d'un objet, ainsi on peut facilement savoir si deux variables sont des *alias*, c'est-à-dire si elles référencent le même objet :

```
>>> fable = ["Je", "plie", "mais", "ne", "romps", "point"]
>>> phrase = fable       # on vient de créer un alias pour la liste
>>> id(phrase)           # l'identifiant de 'phrase'...
139680634898824          # ...est le même que celle de 'fable'
>>> id(fable)
139680634898824
>>> phrase[4] = "casse"  # on modifie phrase...
>>> print(fable)         # ... fable est aussi modifié
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Copie « de surface » *vs* copie « en profondeur »

Si on veut pouvoir effectuer des modifications séparées, il faut affecter l'autre variable par une copie distincte de l'objet, soit en créant une nouvelle séquence dans les cas simples, soit en utilisant le module `copy` dans les cas les plus généraux (autres conteneurs). Si l'on veut aussi que chaque élément et attribut de l'objet soit copié séparément et de façon récursive, on emploie la fonction `copy.deepcopy` :

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = a              # une référence alias
>>> b.append(4)
>>> a
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

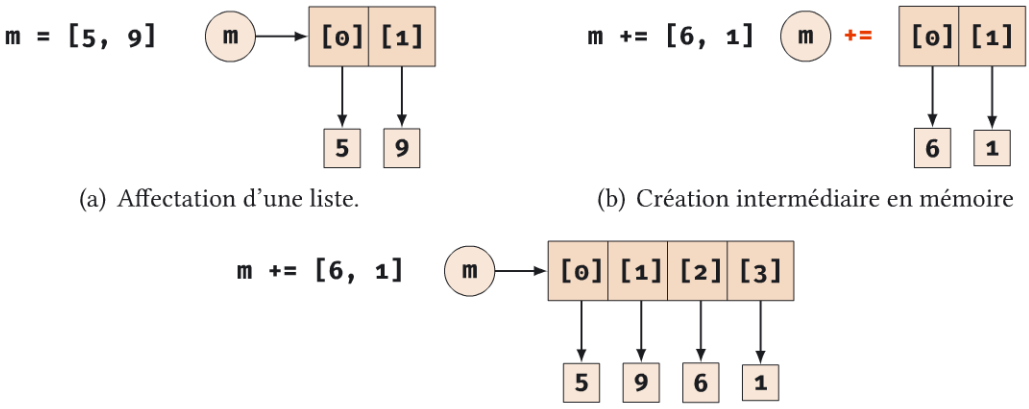
```
[1, 2, 3, 4]
>>> c = a[:]          # une copie simple par extraction de tranche... du début à la fin
>>> c.append(5)
>>> c
[1, 2, 3, 4, 5]
>>> d = copy.copy(a)  # une copie "de surface"
>>> d.append(6)
>>> d
[1, 2, 3, 4, 6]
>>> a
[1, 2, 3, 4]
>>> e = list(a)       # une copie par constructeur
>>> e.append(7)
>>> e
[1, 2, 3, 4, 7]
>>> a
[1, 2, 3, 4]
```

Complément graphique sur l'affectation

Sur les deux figures suivantes, on a représenté d'une part l'affectation augmentée d'un objet **non modifiable** (cas d'un entier : ☞ Fig. 4.1) et d'autre part l'affectation augmentée d'un objet **modifiable** (cas d'une liste : ☞ Fig. 4.2). Dans les deux situations on a représenté l'étape intermédiaire :

(a) Affectation d'un entier (b) Affectation augmentée

FIGURE 4.1 – Affectation augmentée d'un objet non modifiable



(a) Affectation d'une liste. (b) Création intermédiaire en mémoire

(c) Affectation augmentée

FIGURE 4.2 – Affectation augmentée d'un objet modifiable

Copyright © 2017 Dunod.

4.6 Dictionnaires (dict), exemples de tableaux associatifs

Définition

✍ Un **tableau associatif** [1] est un conteneur non ordonné d'éléments indexés par des clés, avec un accès très rapide à un élément à partir de sa clé, chaque clé ne pouvant être présente qu'une seule fois dans la collection.

Un tableau associatif est un genre d'application mathématique entre un ensemble de clés et un ensemble de valeurs.

Python propose le type standard `dict`.

Les dictionnaires constituent un type composite, mais ils n'appartiennent pas aux séquences.

Remarque

✓ On trouvera en annexe F p. 212 une liste complète des opérations et des méthodes sur les dictionnaires.

Définition

✍ Un **dictionnaire** est un tableau associatif modifiable.

Il permet de stocker des couples (ou paires) `(clé, valeur)` avec des valeurs de tous types, éventuellement hétérogènes, et des clés ayant comme contrainte d'être *hachables* [2].

Syntaxe

✎ Couples notés `clé : valeur`, séparés les uns des autres par des virgules et entourés d'accolades.

Les dictionnaires sont *modifiables* mais *non ordonnés* : les couples enregistrés n'occupent pas un ordre immuable, leur emplacement est géré par un algorithme spécifique. Le caractère non ordonné des dictionnaires est le prix à payer pour leur rapidité !

Une *clé* pourra être alphabétique, numérique… : en fait tout type *hachable* convient (donc liste et dictionnaire exclus). Les *valeurs* pourront être de tout type sans exclusion.

Exemples de création

```
>>> d1 = {}                                  # dictionnaire vide. Autre notation : d1 = dict()
>>> d1["nom"] = 3                            # la clé "nom" reçoit la valeur 3
>>> d1["taille"] = 176
>>> d1
{'nom': 3, 'taille': 176}
>>> d2 = {"nom": 3, "taille": 176}           # définition en extension des couples (clé:valeur)
>>> d2
{'nom': 3, 'taille': 176}
>>> d3 = dict(nom=3, taille=176)             # utilisation de paramètres nommés
                                             # (syntaxe d'appel de fonction)
>>> d3
{'taille': 176, 'nom': 3}
>>> d4 = dict([("nom", 3), ("taille", 176)]) # utilisation d'une liste de couples clés/valeurs
>>> d4
{'nom': 3, 'taille': 176}
```

1. En anglais *associative array*. On trouve aussi dans la littérature ou dans d'autres langages le terme *hash map*.
2. Dont les valeurs permettent de calculer une valeur entière – valeur de *hash* – qui ne change pas au cours du temps. C'est cette valeur qui permet ensuite un accès très rapide à partir de la clé *via* un index dans une « table de hachage ».

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Méthodes

Voici quelques méthodes applicables aux dictionnaires :

```
>>> tel = {'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel
{'guido': 4127, 'jack': 4098, 'sape': 4139}  # un dictionnaire n'est pas ordonné
>>> tel['jack']                              # valeur de la clé 'jack'
4098
>>> del tel['sape']                          # suppression d'un coupe (clé : valeur)
>>> tel.keys()                               # clés de tel
dict_keys(['jack', 'guido'])
>>> tel.values()                             # valeurs de tel
dict_values([4098, 4127])
>>> 'guido' in tel, 'jack' not in tel        # teste l'appartenance d'une clé au dictionnaire
(True, False)
```

4.7 Ensembles (set)

Remarque

✓ On trouvera en annexe F p. 213 une liste complète des opérations et des méthodes sur les ensembles.

Définition

✍ Un **ensemble** est une collection itérable non ordonnée d'éléments *hachables* uniques.

Syntaxe

✎ Valeurs séparées les unes des autres par des virgules et entourées d'accolades.

Un `set` est la transposition informatique de la notion d'ensemble mathématique.

En Python, il existe deux types d'ensembles : les ensembles modifiables (`set`) et les ensembles non modifiables (`frozenset`). On retrouve ici les mêmes différences qu'entre les listes et les tuples.

Exemples de construction d'ensembles :

```
>>> couleurs = {'trefle', 'carreau', 'coeur', 'pique'}  # expression littérale
>>> chiffres = set(range(10))        # construction à partir des éléments d'un itérable
>>> couleurs
{'coeur', 'trefle', 'pique', 'carreau'}
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Exemples d'opérations sur les ensembles :

```
X = set('abcdef')   # X = {'a', 'b', 'c', 'd', 'e', 'f'}
Y = set('efghf')    # Y = {'e', 'f', 'g', 'h'} pas de duplication : qu'un seul 'f'
'b' in X, 'c' in Y  # (True, False)
X - Y  # {'a', 'b', 'c', 'd'} ensemble des éléments de X qui ne sont pas dans Y
Y - X  # {'g', 'h'} ensemble des éléments de Y qui ne sont pas dans X
X | Y  # {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'} union
X & Y  # {'e', 'f'} intersection
X ^ Y  # {'a', 'b', 'c', 'd', 'g', 'h'} ensemble des éléments qui sont soit dans X, soit dans Y
```

Copyright © 2017 Dunod.

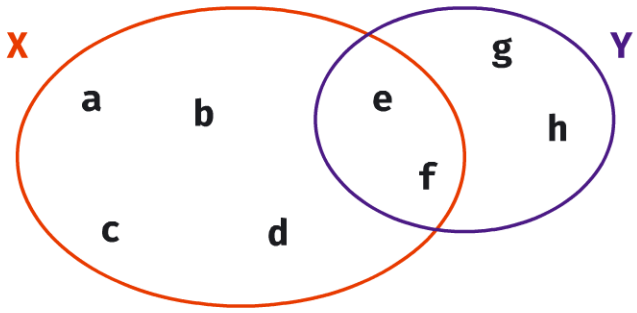


FIGURE 4.3 – Opérations sur les ensembles.

4.8 Itérer sur les conteneurs

Les techniques suivantes sont classiques et très utiles.

Obtenir clés et valeurs en bouclant sur un dictionnaire

```
knights = {"Gallahad": "the pure", "Robin": "the brave"}
for k, v in knights.items():
    print(k, v)
# Gallahad the pure
# Robin the brave
```

Obtenir indice et élément en bouclant sur une liste

```
>>> for i, v in enumerate(["tic", "tac", "toe"]):
...     print(i, '->', v)
...
0 -> tic
1 -> tac
2 -> toe
```

Boucler sur deux séquences (ou plus) appariées

La fonction `zip()` fait allusion à la fermeture Éclair qui joint et entrelace deux rangées de dents. Elle permet de fournir à chaque itération les valeurs de même index issues de plusieurs séquences.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['Lancelot', 'the Holy Grail', 'blue']
>>> for question, answer in zip(questions, answers):
...     print('What is your', question, '? It is', answer)
...
What is your name ? It is Lancelot
What is your quest ? It is the Holy Grail
What is your favorite color ? It is blue
```

Obtenir une séquence inversée (la séquence initiale est inchangée)

```
for i in reversed(range(1, 10, 2)):
    print(i, end=" ")       # 9 7 5 3 1
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Obtenir une séquence triée (la séquence initiale est inchangée)

```
basket = ["apple", "orange", "apple", "pear", "orange", "banana"]
for f in sorted(basket):
    print(f, end=" ")   # apple apple banana orange pear
```

Obtenir une séquence triée à éléments uniques (la séquence initiale est inchangée)

```
basket = ["apple", "orange", "apple", "pear", "orange", "banana"]
for f in sorted(set(basket)):
    print(f, end=" ")   # apple banana orange pear
```

4.9 Exercices

1. ✠ Le *mélange de Monge* d'un paquet de cartes numérotées de 2 à $2n$ consiste à démarrer un nouveau paquet avec la carte 1, à placer la carte 2 au-dessus de ce nouveau paquet, puis la carte 3 au-dessous du nouveau paquet et ainsi de suite en plaçant les cartes paires au-dessus du nouveau paquet et les cartes impaires au-dessous.
   Écrire un script qui affiche le paquet initial et le paquet mélangé.
   Exemple d'affichage pour $n = 5$:

   ```
   Paquet initial   : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   Mélange de Monge : [10, 8, 6, 4, 2, 1, 3, 5, 7, 9]
   ```

2. L'utilisateur saisit un entier n entre 2 et 12, le programme donne le nombre de façons de faire n en lançant deux dés.

3. Même problème que le précédent mais avec n entre 3 et 18 et trois dés.

4. ✠✠ Généralisation des deux questions précédentes. L'utilisateur saisit deux entrées, d'une part le nombre de dés, nbd (que l'on limitera pratiquement à 10), et d'autre part la somme, s, comprise entre nbd et $6nbd$. Le programme calcule et affiche le nombre de façons de faire s avec les nbd dés.
   Exemple d'exécution :

   ```
   Nombre de dés [2 .. 8] : 6
   Entrez un entier [6 .. 36] : 21
   Il y a 4332 façons de faire 21 avec 6 dés.
   ```

Copyright © 2017 Dunod.

Chapitre 5

Fichiers textuels

L'ordinateur n'exécute que les programmes présents dans sa mémoire volatile (la RAM). Pour conserver durablement des informations, il faut utiliser une mémoire permanente (un disque dur, une clé USB, un DVD) sur laquelle le système d'exploitation organise les données sous la forme de fichiers.

Nous limiterons nos exemples aux fichiers *textuels* (lisibles par un éditeur), mais signalons que les fichiers enregistrés en codage *binaire* permettent de stocker de nombreuses autres catégories de données d'une façon compacte et efficace. Cependant, l'accès aux fichiers binaires est généralement plus complexe et il est conseillé d'utiliser des modules Python spécifiques qui masquent cette complexité.

5.1 Gestion des fichiers

L'ouverture d'un fichier est réalisée en utilisant la fonction standard `open()`, qui prend en premier paramètre une chaîne de caractères indiquant le chemin d'accès et le nom du fichier, en deuxième paramètre une chaîne de caractères indiquant un mode d'ouverture, et en troisième paramètre (optionnel mais recommandé) une indication d'encodage [cf. p. 177] pour les caractères dans le fichier.

Nommage des fichiers

Remarque : Les noms des fichiers et des répertoires doivent respecter les règles définies au niveau du système d'exploitation, qui peuvent varier d'un système à l'autre. On évitera en général les caractères `/ * ? < > " | :`.

Copyright © 2017 Dunod.

Sans spécification de chemin d'accès, les fichiers sont ouverts dans le répertoire courant (*current working directory*), qui peut être le répertoire qui contient le script Python exécuté (typiquement lorsqu'on travaille avec `Pyzo`), ou le répertoire utilisé lors du lancement du script (typiquement lorsqu'on démarre un script *via* une console), ou encore tout autre répertoire après que le répertoire courant a été modifié par l'utilisateur ou par le programme [1].

Le chemin d'accès est constitué d'une série de noms de répertoires à traverser pour accéder au fichier ; un séparateur (`/` sous Linux/MacOS X/Windows et `\` sous Windows) permet de séparer les différents noms. L'origine de ce chemin peut être **absolue** (par rapport à la « racine » de l'arborescence de fichiers sous Linux [2]/MacOS X, la « racine » d'un volume disque sous Windows), ou bien **relative** (par rapport au répertoire courant). Lors du parcours des répertoires pour atteindre un fichier, le répertoire spécial `.` correspond au répertoire actuel dans le parcours, et le répertoire spécial `..` correspond au répertoire parent du répertoire actuel dans le parcours ; ceci permet de remonter dans l'arborescence et de réaliser des parcours relatifs.

Remarque : L'utilisation du séparateur `\` entre les noms dans les chemins sous Windows est un piège lorsqu'on exprime ces chemins dans les programmes. On utilise en effet des chaînes de caractères, et `\` est le caractère d'échappement dans ces chaînes (`\t` pour tabulation, `\n` pour retour à la ligne… [cf. p. 25]). En Python, il y a plusieurs façons d'éviter ce problème sous Windows :

- doubler tous les `\` : `"C:\\Users\\Moi\\Documents\\mon_fichier.txt"` ;
- utiliser des chaînes littérales brutes (*raw string*) en les préfixant par un `r` : `r"C:\Users\Moi\Documents\mon_fichier.txt"` ;
- utiliser le séparateur `/` comme sous Linux (ce que permet Windows) : `"C:/Users/Moi/Documents/mon_fichier.txt"` ;
- utiliser le module `os.path`. Il propose des fonctions qui se chargent d'insérer le bon séparateur quelle que soit la plateforme utilisée, comme présenté ci-après [cf. p. 56]. Signalons également la classe `Path` du module `pathlib`.

5.1.1 Ouverture et fermeture des fichiers

```
f1 = open("monFichier_1", "r", encoding='utf8')  # "r" mode lecture (par défaut)
f2 = open("monFichier_2", "w", encoding='utf8')  # "w" mode écriture (à partir d'un fichier vide)
f3 = open("monFichier_3", "a", encoding='utf8')  # "a" mode ajout (concaténation en écriture)
```

Python ouvre les fichiers en mode *texte* par défaut (mode `"t"`). Pour les fichiers *binaires*, il faut préciser explicitement le mode `"b"` (par exemple : `"wb"` pour une écriture en mode binaire).

Encodage des caractères

Le paramètre optionnel `encoding` assure les conversions entre les types `byte` (c'est-à-dire des tableaux d'octets), format de stockage des fichiers sur le disque, et le type `str` (qui, en Python 3, signifie toujours Unicode), manipulé lors des lectures et écritures. Il est prudent de toujours le spécifier pour les fichiers textuels (cela oblige à se poser la question de l'encodage).

Les encodages [3] les plus fréquents sont `'utf8'` (c'est l'encodage à privilégier en Python 3), `'latin1'` (format par défaut des fichiers `html`), `'ascii'`… L'utilisation du mauvais encodage peut faire apparaître ce genre de « bogues » [4] que vous avez sûrement déjà vus sur des pages web [cf. p. 179].

1. Voir les fonctions `getcwd()` et `chdir()` du module standard `os`.
2. Ou autre système type Unix comme l'est GNU/Linux.
3. Cf. annexe C p. 177
4. Ou *bugs*.

Copyright © 2017 Dunod.

Veillez à la bonne fermeture des fichiers !

Tant que le fichier n'est pas fermé [1], son contenu n'est pas garanti sur le disque. En effet, le système d'exploitation ainsi que les bibliothèques intermédiaires d'accès aux fichiers utilisent des « espaces tampons » en mémoire RAM pour travailler efficacement, et ces espaces ne sont pas écrits systématiquement immédiatement ; un crash violent d'un programme ou du système complet peut faire perdre des données qui n'auraient pas été physiquement écrites sur disque.

```
f1.close()   # Une seule méthode de fermeture
```

Dans le chapitre dédié aux techniques avancées de Python [cf. p. 121], nous verrons une syntaxe d'écriture qui permet à coup sûr de ne pas oublier de fermer ses fichiers !

5.1.2 Écriture séquentielle

Le fichier sur disque est considéré comme une séquence de caractères qui sont ajoutés à la suite, au fur et à mesure que l'on écrit dans le fichier.

Méthodes d'écriture :

```
f = open("truc.txt", "w", encoding='utf8')
s = 'toto\n'
f.write(s)         # écrit la chaîne s dans f
l = ['a', 'b', 'c']
f.writelines(l)    # écrit les chaînes de la liste l dans f
f.close()

f2 = open("truc2.txt", "w", encoding='utf8')
print("abcd", file=f2)  # utilisation de l'option file avec 'print'
f2.close()
```

Ce qui produit les enregistrements suivants :

```
Fichier truc.txt :

toto
abc

Fichier truc2.txt :

abcd
```

5.1.3 Lecture séquentielle

En lecture, la séquence de caractères qui constitue le fichier est parcourue en commençant au début du fichier et en avançant au fur et à mesure des lectures.

Méthodes de lecture en mémoire d'un fichier en entier :

```
f = open("truc.txt", "r", encoding='utf8')
s = f.read()           # lit tout le fichier --> chaîne
f.close()
```

1. Ou bien *flushé* par un appel à la méthode `flush()`.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
f = open("truc.txt", "r", encoding='utf8')
s = f.readlines()      # lit tout le fichier --> liste de chaînes
f.close()
```

Méthodes de lecture d'une partie d'un fichier [1] :

```
f = open("truc.txt", "r", encoding='utf8')
s = f.read(3)                          # lit au plus n octets --> chaîne
s = f.readline()                       # lit la ligne suivante --> chaîne
f.close()

# Affichage des lignes d'un fichier une à une
f = open("truc.txt", encoding='utf8')  # mode "r" par défaut
for ligne in f:
    print(ligne)
f.close()
```

5.2 Travailler avec des fichiers et des répertoires

Dès que l'on manipule les répertoires [cf. p. 54], on a besoin de se déplacer dans l'arborescence des fichiers, de connaître les noms de base ou l'extension de leur nom, etc.

5.2.1 Se positionner dans l'arborescence

```
>>> import os
>>> os.chdir('/home/bob/Esperanto')  # changer de répertoire
>>> os.getcwd()                      # connaître le répertoire courant
'/home/bob/Esperanto'
```

5.2.2 Construction de noms de chemins

```
>>> import os.path
>>> os.path.join('home', 'bob/Esperanto', 'Baza_kurso')  # concaténer les noms d'un chemin
'home/bob/Esperanto/Baza_kurso'
>>> os.path.join('/home', 'bob', 'Esperanto', 'Baza_kurso')
'/home/bob/Esperanto/Baza_kurso'
>>> os.path.expanduser('~/Esperanto')   # remplace ~ par le 'home' de l'utilisateur
'/home/bob/Esperanto'
>>> os.path.join(os.path.expanduser('~'), 'Esperanto')
'/home/bob/Esperanto'
```

5.2.3 Opérations sur les noms de chemins

```
>>> import os.path
>>> os.path.exists('/home/bob/Esperanto/Inconnu')
False
>>> os.path.exists('/home/bob/Esperanto/Brassens')  # c'est un répertoire
True
>>> os.path.isfile('/home/bob/Esperanto/brassens')  # l'OS Linux est sensible à la casse !
```

1. Nous ne détaillerons pas plus les méthodes des fichiers, sachez qu'il est possible de connaître et de modifier la position de lecture/écriture dans un fichier (méthodes `tell()` et `seek()`), ainsi que de « retailler » un fichier à une taille donnée (méthode `truncate()`).

Copyright © 2017 Dunod.

```
False
>>> os.path.isfile('/home/bob/Esperanto/Brassens/Brassens-eo.odt')
True
>>> os.path.dirname('/home/bob/Esperanto')
'/home/bob'
>>> os.path.basename('/home/bob/Esperanto')
'Esperanto'
>>> os.path.split('/home/bob/Esperanto')            # retourne le tuple (dirname, basename)
('/home/bob', 'Esperanto')
>>> (shortname, extension) = os.path.splitext('gerda.pdf')
>>> shortname, extension
('gerda', '.pdf')
```

5.2.4 Gestion d'un répertoire

```
>>> import os
>>> os.listdir('/home/bob/Esperanto/Brassens')
['Brassens-fr.pdf', 'Brassens-eo.odt', 'Brassens-fr.odt']
>>> os.mkdir("/home/bob/un")
>>> os.makedirs("/home/bob/un/sous/repertoire/ici/et/la")
>>> os.rmdir("/home/bob/un/sous/repertoire/ici/et/la")
>>> os.rmdir("/home/bob/un/sous/repertoire/ici/et")
>>> os.rmdir("/home/bob/un/sous/repertoire/ici")
```

Signalons également le module standard `shutil`, qui autorise des opérations de haut niveau sur des arborescences de répertoires et de fichiers comme la copie, la suppression ou le renommage.

5.3 Exercices

1. Soit une liste `lst` contenant des tuples de valeurs `(nom, masse, volume)`, où `nom` est le nom d'un élément chimique, `masse` est une valeur flottante exprimée en grammes et `volume` une valeur flottante exprimée en cm³.

   ```
   lst = [('Arsenic', 17.8464, 3.12), ('Aluminium', 16.767, 6.21), ('Or', 239320, 12400)]
   ```

   Enregistrer les données de `lst` dans un fichier texte, dont chaque ligne correspond à un élément, avec le format suivant : nom de l'élément = masse volumique g/cm3.

   Par exemple :

   ```
   Arsenic = 5.72 g/cm3
   ```

2. Soit un fichier texte `elements.txt` contenant :

   ```
   Arsenic = 5.72 g/cm3
   Aluminium = 2.70 g/cm3
   Or = 19.30 g/cm3
   ```

   Écrire un programme qui demande à un utilisateur de saisir un volume en cm³, puis qui utilise le fichier ci-dessus afin d'afficher pour chaque élément la masse correspondant au volume saisi.

3. Le module `time` fournit une fonction `asctime()` retournant sous forme de chaîne la date et l'heure courantes formatées. Écrire une fonction `note_journal(nomfichier, message)` qui à chaque appel **ajoute à la fin du fichier**, dont le nom est donné en paramètre, une ligne contenant la date et l'heure courantes suivies du message donné lui aussi en paramètre.

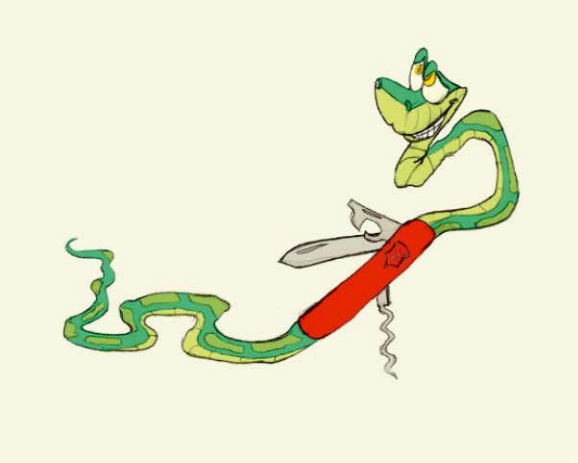
Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Note : Pour les tests, il est possible de faire des pauses dans le programme avec la fonction `sleep(n)`, où `n` est un nombre flottant de secondes, du module `time`

Copyright © 2017 Dunod.

Chapitre 6

Fonctions et espaces de noms



Les fonctions sont les éléments structurants de base de tout langage procédural.

Elles offrent différents avantages. Elles évitent la répétition (on peut « factoriser » une portion de code qui se répète lors de l'exécution en séquence d'un script), elles mettent en relief les données et les résultats (entrées et sorties de la fonction), elles permettent la réutilisation (mécanisme de l'*import*), enfin elles décomposent une tâche complexe en tâches plus simples (conception de l'application).

6.1 Définition et syntaxe

Nous avons déjà rencontré des fonctions internes à Python (appelées *builtin*), par exemple la fonction `len()`. Intéressons-nous maintenant aux fonctions définies par l'utilisateur.

Définition

✍ Une **fonction** est un ensemble d'instructions regroupées sous un *nom* et s'exécutant à la demande (l'*appel* de la fonction).

On doit définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; il s'agit d'une « factorisation de code ».

Syntaxe

✎ La définition d'une fonction se compose :

- du **mot clé** `def` suivi de l'identificateur de la fonction, de parenthèses entourant les paramètres de la fonction séparés par des virgules, et du caractère « deux-points » qui termine toujours une instruction composée (c'est l'*en-tête* de la fonction) ;
- d'une **chaîne de documentation** (ou *docstring*) indentée comme le corps de la fonction ;

Copyright © 2017 Dunod.

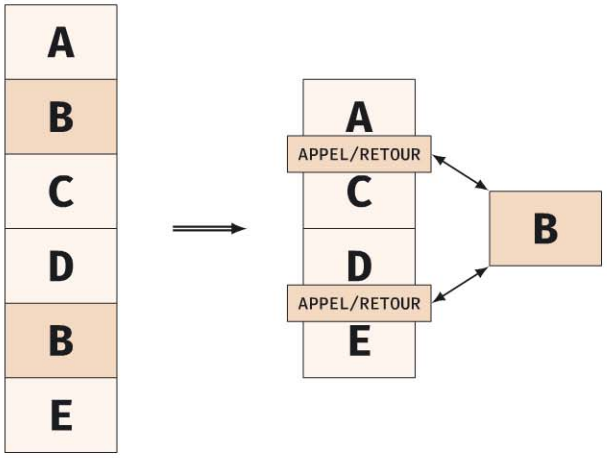
— du **bloc d'instructions indenté** par rapport à la ligne de définition, et qui constitue le corps de la fonction.

Le bloc d'instructions est *obligatoire*. S'il est vide, on emploie l'instruction `pass`.

La documentation, bien que facultative, est *fortement* conseillée [1].

```
def volumeEllipsoide(a, b, c):
    """Retourne le volume d'un ellipsoïde de demi-grand axes a, b et c."""
    return  3.14 * a * b * c * 4 / 3

def fonctionVide():
    """Une fonction sans corps doit contenir l'instruction 'pass'."""
    pass
```

L'utilisation des fonctions évite la duplication du code :



Les fonctions mettent en relief les entrées et les sorties :

```
# util.py
def proportion(chaine, motif) :
    """Fréquence de <motif> dans <chaine>."""
    n = len(chaine)
    k = chaine.count(motif)
    return k/n
```

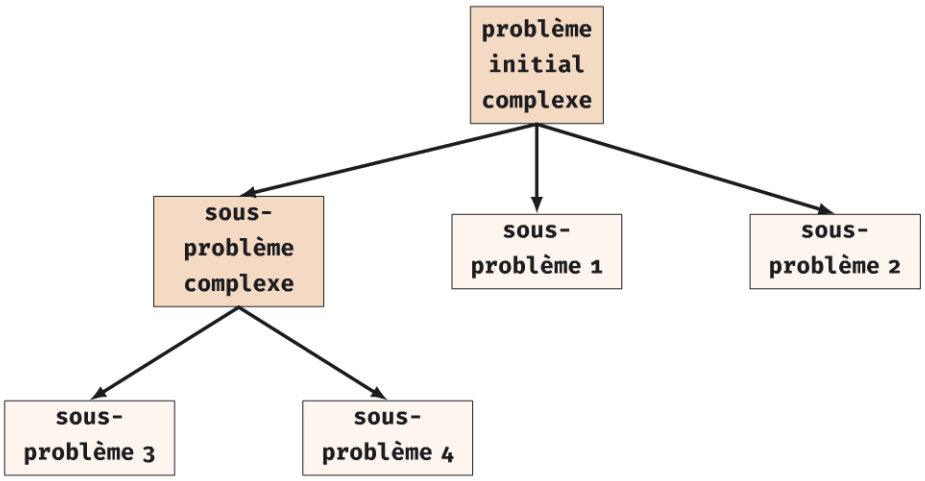
L'instruction import permet la réutilisation du code défini dans d'autres fichiers :

```
import util
...
p1 = util.proportion(une_chaine, 'le')
...
p2 = util.proportion(une_autre_chaine, 'des')
...
```

1. La documentation des sources sera revue plus en détail [cf. p. 145].

Copyright © 2017 Dunod.

L'utilisation des fonctions améliore la conception d'un programme :



6.2 Passage des arguments

La plupart du temps, les fonctions que nous allons définir auront besoin d'informations que nous leur fournirons sous forme d'arguments.

6.2.1 Mécanisme général

Remarque

✓ En Python, les arguments sont passés *par affectation* : chaque argument de l'appel correspond, *dans l'ordre*, à un paramètre de la définition de la fonction. La correspondance se fait par affectation des arguments aux paramètres.

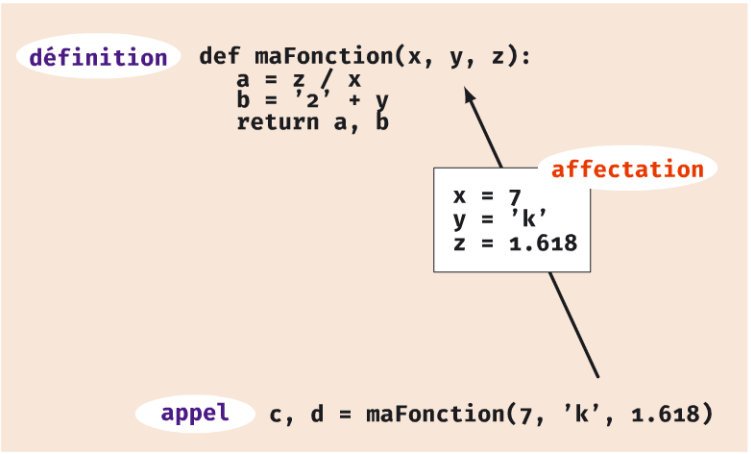


FIGURE 6.1 – Passage par affectation des arguments d'appel aux paramètres de définition

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

6.2.2 Un ou plusieurs paramètres, pas de retour

Exemple sans l'instruction `return`, ce qu'on appelle souvent une « procédure » [1]. Dans ce cas, la fonction renvoie implicitement la valeur `None` [2] :

```
def table(base, debut, fin):
    """Affiche la table de multiplication des <base> de <debut> à <fin>."""
    n = debut
    while n <= fin:
        print(n, 'x', base, '=', n * base)
        n += 1

# exemple d'appel :
table(7, 2, 8)
# 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49 8 x 7 = 56

# autre exemple du même appel, mais en nommant les paramètres dont on peut changer l'ordre ;
table(debut=2, fin=8, base=7)
```

6.2.3 Un ou plusieurs paramètres, un ou plusieurs retours

Exemple avec utilisation d'un `return` d'une valeur unique. Le résultat de l'évaluation de la fonction peut être utilisé dans une expression ou stocké dans une variable :

```
from math import pi

def cube(x):
    """Retourne le cube de l'argument."""
    return x**3

def volumeSphere(r):
    """Retourne le volume d'une sphère de rayon <r>;"""
    return 4.0 * pi * cube(r) / 3.0

# Saisie du rayon et affichage du volume
rayon = float(input('Rayon : '))
print("Volume de la sphère =", volumeSphere(rayon))
```

Exemple avec utilisation d'un `return` de multiples valeurs. Les résultats sont retournés par Python dans un tuple. Celui-ci peut être « décapsulé » dans un nombre correspondant de variables, ou bien stocké dans une seule variable en tant que tuple (dont les éléments pourront ultérieurement être accédés par leur indice) :

```
import math

def surfaceVolumeSphere(r):
    surf = 4.0 * math.pi * r**2
    vol = surf * r/3
    return surf, vol

# programme principal
rayon = float(input('Rayon : '))
```

1. Une fonction *vaut* quelque chose (son retour), une procédure *fait* quelque chose.
2. On a parfaitement le droit de coder explicitement `return None`.

Copyright © 2017 Dunod.

```
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface {:g} et de volume {:g}".format(s, v))
```

Attention

L'instruction `return` fait immédiatement sortir du flux des instructions de la fonction.

6.2.4 Passage d'une fonction en paramètre

En Python, une fonction est un objet dit de « premier ordre » (ou de « première classe »), manipulable comme toute valeur, ce qui signifie entre autres qu'une variable peut référencer une fonction. On peut donc transmettre une fonction comme paramètre et on peut retourner une fonction.

```
>>> def f(x):
...     return 2*x+1
...
>>> def g(x):
...     return x//2
...
>>> def h(fonc, x):
...     return fonc(x)
...
>>> h(f, 3)
7
>>> h(g, 4)
2
```

6.2.5 Paramètres avec valeur par défaut

Il est possible de spécifier, lors de la déclaration, des valeurs par défaut à utiliser pour les arguments. Cela permet, lors de l'appel, de ne pas avoir à spécifier les paramètres correspondants.

Il est également possible, en combinant les valeurs par défaut et le nommage des paramètres, de n'indiquer à l'appel que les paramètres dont on désire modifier la valeur de l'argument. Il est par contre nécessaire, lors de la définition, de regrouper tous les paramètres optionnels avec leurs valeurs par défaut à la fin de la liste des paramètres.

```
>>> def accueil(nom, prenom, depart="MP", semestre="S2"):
...     print(prenom, nom, "Département", depart, "semestre", semestre)
...
>>> accueil("Deuf", "John")
John Deuf Département MP semestre S2
>>> accueil("Paradise", "Eve", "Info")
Eve Paradise Département Info semestre S2
>>> accueil("Annie", "Steph", semestre="S3")
Steph Annie Département MP semestre S3
```

Attention

On utilise de préférence des valeurs par défaut *non modifiables* (types `int`, `float`, `str`, `bool`, `tuple` ...) car la modification d'un paramètre par un premier appel est visible les fois suivantes (« effet de bord » [cf. p. 65]).

Si on a besoin d'une valeur par défaut qui soit *modifiable* (`list`, `dict`), on utilise la valeur prédéfinie `None` et on fait un test dans la fonction pour mettre en place la valeur par défaut :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
def maFonction(liste=None):
    if liste is None:
        liste = [1, 3]
```

6.2.6 Nombre d'arguments arbitraire : passage d'un tuple de valeurs

Le passage d'un nombre arbitraire d'arguments est permis en utilisant la notation d'un argument final `*nom`. Les paramètres surnuméraires sont alors transmis sous la forme d'un tuple affecté à cet argument (que l'on appelle généralement `args`).

```
def somme(*args):
    """Renvoie la somme du tuple <args>."""
    resultat = 0
    for nombre in args:
        resultat += nombre
    return resultat

# Exemples d'appel :
print(somme(23))            # 23
print(somme(23, 42, 13))    # 78
```

Remarque

✓ Le paramètre `*` peut être défini sans nom, auquel cas il doit être suivi de paramètres nommés avec des valeurs par défaut. Les valeurs pour ces paramètres nommés ne peuvent alors être fournies qu'en nommant explicitement chaque argument.

Réciproquement, il est aussi possible de passer un tuple (en fait une séquence) à l'appel, qui sera *décapsulé* en une liste de paramètres d'une fonction « classique ».

```
def somme(a, b, c):
    return a+b+c

# Exemple d'appel :
elements = (2, 4, 6)
print(somme(*elements))     # 12
```

6.2.7 Nombre d'arguments arbitraire : passage d'un dictionnaire

De la même façon, il est possible d'autoriser le passage d'un nombre arbitraire d'arguments nommés en plus de ceux prévus lors de la définition en utilisant la notation d'un argument final `**nom`. Les paramètres surnuméraires nommés sont alors transmis sous la forme d'un dictionnaire affecté à cet argument (que l'on appelle généralement `kwargs` pour *keyword args*).

Réciproquement, il est aussi possible de passer un dictionnaire à l'appel d'une fonction, qui sera décapsulé, chaque clé étant liée au paramètre correspondant de la fonction.

```
def unDict(**kwargs):
    return kwargs

# Exemples d'appels
## par des paramètres nommés :
print(unDict(a=23, b=42))      # {'a': 23, 'b': 42}
```

Copyright © 2017 Dunod.

```
## en fournissant un dictionnaire :
mots = {'d': 85, 'e': 14, 'f':9}
print(unDict(**mots))          # {'e': 14, 'd': 85, 'f': 9}
```

Attention

☢ Si la fonction possède plusieurs arguments, le dictionnaire est en *toute dernière* position (après un éventuel `*nom`).

6.2.8 Argument modifiable (ou *mutable*)

Lorsque l'on passe à une fonction un argument non modifiable [1] (entier, chaîne…), il peut être utilisé sans restriction et sans avoir à se poser de question. Par contre, lorsque l'on passe à une fonction un argument modifiable (liste, dictionnaire…), alors il faut avoir conscience que toute modification sur celui-ci dans la fonction persistera après la sortie de la fonction, on appelle cela un « effet de bord » car la fonction modifie des données qui sont définies hors de sa portée locale (on appelle aussi « effet de bord » le fait de modifier une variable globale).

```
# Opération avec paramètre immutable
def additionne_1(x):
    x = x + 1
    return x

a = 3

print(additionne_1(a))  # 4
print(a)                # 3         (a n'est pas modifié)

# Opération avec paramètre mutable
def ajoute_1(x):
    x.append(1)
    return x

lst = [1, 4, 5]

print(ajoute_1(lst))      # [1, 4, 5, 1]
print(lst)                # [1, 4, 5, 1]    (lst est modifié à chaque appel)
print(ajoute_1(lst))      # [1, 4, 5, 1, 1]
print(lst)                # [1, 4, 5, 1, 1]
print(ajoute_1(lst))      # [1, 4, 5, 1, 1, 1]
print(lst)                # [1, 4, 5, 1, 1, 1]
```

Note : C'est cette possibilité d'effet de bord sur les paramètres mutables qui explique l'encart Attention du paragraphe 6.2.5. Si vous définissez un paramètre avec une valeur par défaut mutable, soyez conscient des implications (mémoire des effets de bord sur la valeur par défaut fournie à la définition, qui est reprise à chaque appel).

6.3 Espaces de noms

Un espace de noms est une notion permettant de lever une ambiguïté sur des termes qui pourraient être *homonymes* sans cela. Il est matérialisé par un préfixe identifiant de manière unique l'origine d'un terme. Au sein d'un même espace de noms, il n'y a pas d'homonymes. Dans l'exemple

1. Ou *immutable*

© Dunod. Toute reproduction non autorisée est un délit.
Copyright © 2017 Dunod.

suivant, les fonctions `open()` ne sont pas homonymes car elles appartiennent à des packages différents :

```
>>> import webbrowser, os, PIL.Image
>>> webbrowser.open("http://www.dunod.fr")
True
>>> os.open("/etc/hosts", os.O_RDONLY)
4
>>> PIL.Image.open("../figs/chap5_r.png")
<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=366x519 at 0x7F9D65060E10>
```

Nous verrons dans les chapitres suivants que cette notion d'espaces de noms avec la notation pointée est centrale en Python et se retrouve en bien d'autres endroits (modules, classes, objets...).

6.3.1 Portée des objets

On distingue :

- la portée **globale** du module ou du fichier script en cours. L'instruction `globals()` fournit un dictionnaire contenant les couples `(nom, valeur)` de portée globale dans l'espace de noms courant ;
- la portée **locale** des objets internes aux fonctions, des paramètres et des variables affectées dans les fonctions. Tous ces objets sont locaux, leur durée de vie est liée à l'appel courant de la fonction ; si aucune référence à ces objets n'est maintenue après l'appel de la fonction [1], alors ils disparaissent. Les objets globaux ne peuvent pas être réaffectés dans les portées locales sans une directive spécifique (voir exemple suivant). L'instruction `locals()` fournit un dictionnaire contenant les couples `(nom, valeur)` de portée locale dans l'espace de noms courant.

6.3.2 Résolution des noms : règle « LGI »

La recherche des noms est d'abord locale (`L`), puis globale (`G`), enfin interne (`I`) (☞ Fig. 6.2) :

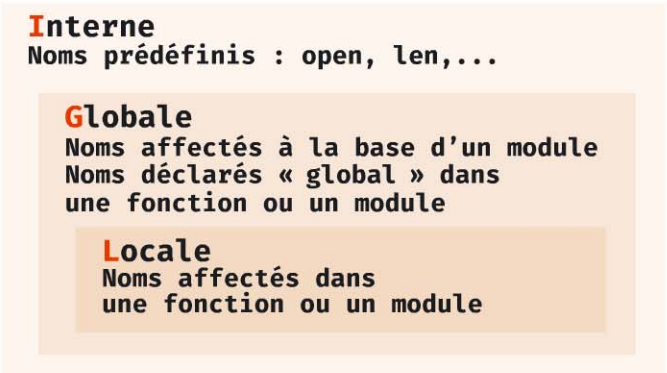


FIGURE 6.2 – Règle LGI

1. Par exemple par un retour de valeur ou par un stockage dans un espace persistant après l'appel de la fonction.

Copyright © 2017 Dunod.

Exemples de portée

Par défaut, tout identificateur **affecté** dans le corps d'une fonction est local à celle-ci. Si une fonction a besoin de réaffecter certains identificateurs globaux, la première instruction de cette fonction doit être : `global <identificateur>`.

Par exemple :

```
# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def f1(v):
    global portee
    portee = 'modifiée dans f1()'
    return v + portee

def f2(v):
    return v + portee

def f3(v):
    portee = 'locale à f3(), je masque la portée du module'
    return v + portee

# Programme principal ========================================================
x, portee = 'Je suis ', 'globale au module'
print(f1(x))            # Je suis modifiée dans f1()
print(portee)           # modifiée dans f1()

x, portee = 'Je suis ', 'globale au module'
print(f2(x))            # Je suis globale au module
print(portee)           # globale au module

x, portee = 'Je suis ', 'globale au module'
print(f3(x))            # Je suis locale à f3(), je masque la portée du module
print(portee)           # globale au module
```

6.4 Exercices

1. Écrire un programme qui approxime par défaut la valeur de la constante mathématique e, pour n assez grand, en utilisant la formule :

$$e \approx \sum_{i=0}^{n} \frac{1}{i!}$$

   Pour cela, définissez la fonction `factorielle()` et, dans votre programme principal, saisissez l'ordre n et affichez l'approximation correspondante de e.

2. Un gardien de phare logeant au rez-de-chaussée doit monter n fois par jour en haut du phare pour maintenir l'optique.

   Écrire une procédure (donc sans `return`) `hauteurParcourue()` qui reçoit trois paramètres, le nombre de tours de maintenance, le nombre de marches du phare et la hauteur de chaque marche (en cm), et qui affiche :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
Pour n tours de maintenance et x marches de y cm,
il parcourt un dénivelé de z.zz m par semaine.
```

On n'oubliera pas :
— qu'une semaine comporte 7 jours ;
— qu'une fois en haut, le gardien doit redescendre ;
— que le résultat est à exprimer en mètres.

3. Écrire une fonction `minMaxMoy()` qui reçoit une liste d'entiers et qui renvoie un tuple de trois valeurs : le minimum, le maximum et la moyenne de cette liste. La fonction ne doit pas utiliser les fonctions standard `min()`, `max()` et `sum()`.

   Le programme principal appellera cette fonction avec la liste : `[10, 18, 14, 20, 12, 16]`.

Copyright © 2017 Dunod.

Chapitre 7

Modules et packages

Un programme Python est généralement composé de plusieurs fichiers sources, appelés *modules*.

S'ils sont correctement codés, les modules sont indépendants les uns des autres et peuvent être réutilisés à la demande dans d'autres programmes.

Ce chapitre explique comment coder des modules et comment les importer pour les utiliser.

Nous verrons également la notion de *package* qui permet de grouper plusieurs modules.

7.1 Modules

Définition

✍ **Module** : fichier contenant une collection d'outils (fonctions, classes, données) apparentés définissant des éléments de programme réutilisables. On utilise aussi souvent le terme de « bibliothèque ».

L'utilisation des modules est très fréquente. En voici les avantages :

- réutilisation du code ;
- isolation, dans un espace de noms identifié, de fonctionnalités particulières ;
- mise en place de services ou de données partagés.

Par ailleurs :

- la documentation et les tests peuvent être intégrés au module ;
- le mécanisme d'import crée un nouvel espace de noms et exécute toutes les instructions du fichier `.py` associé dans cet espace de noms, ce qui permet de réaliser des initialisations lors du chargement du module.

Copyright © 2017 Dunod.

7.1.1 Import

L'instruction `import` charge et exécute le module indiqué s'il n'est pas déjà chargé. L'ensemble des définitions contenues dans ce module devient alors disponible : variables globales, fonctions, classes.

Suivant la syntaxe utilisée, on accède aux définitions du module de différentes façons :

— l'instruction `import nom_module` donne accès à l'ensemble des définitions du module importé en utilisant le nom du module comme espace de noms ;

```
>>> import tkinter
>>> print("Version de tkinter :", tkinter.TkVersion)
Version de tkinter : 8.5
```

— l'instruction `from nom_module import nom1, nom2...` donne accès directement à une sélection choisie de noms définis dans le module.

```
>>> from math import pi, sin
>>> print("Valeur de Pi :", pi, "sinus(pi/4) :", sin(pi/4))
Valeur de Pi : 3.14159265359 sinus(pi/4) : 0.707106781187
```

Dans les deux cas, le module et ses définitions existent dans leur espace mémoire propre, et on duplique simplement dans le module courant les noms que l'on a choisis, comme si on avait fait les affectations : `sin = math.sin` et `pi = math.pi`.

Attention

☢ La syntaxe `from nom_module import *` permet d'importer directement **tous** les noms du module. Cet usage est à prohiber hors des tests car on ne sait pas quels noms sont importés (risques de masquages), et on perd l'origine des noms dans le module importateur.

Remarque

✓ Lorsqu'on parle d'un module ou qu'on l'importe, on omet son extension. Pour l'apprentissage, on considérera que le module `un_module` est dans le fichier `un_module.py`. Il existe toutefois de nombreux modules Python sous la forme de librairies partagées (`.so, .dll, .dylib`...) contenant du code machine directement exécutable, construites à partir de sources en C ou C++ et qui sont utilisées exactement de la même façon que les modules `.py`.

Il est conseillé d'importer *dans l'ordre* :

— les modules de la bibliothèque standard ;
— les modules des bibliothèques tierces ;
— les modules personnels.

Attention

☢ Pour tout ce qui est fonction et classe, ainsi que pour les « constantes » (variables globales définies et affectées une fois pour toutes à une valeur), l'import direct du nom ne pose pas de problème. Par contre, pour les variables globales que l'on désire pouvoir modifier, il est préconisé de passer systématiquement par l'espace de noms du module afin de s'assurer de l'existence de cette variable en un unique exemplaire ayant la même valeur dans tout le programme.

Copyright © 2017 Dunod.

7.1.2 Localisation des fichiers modules

Pour localiser les fichiers de modules et les charger, Python consulte une liste de chemins à la recherche du module demandé. Cette liste est visible (et modifiable) par l'intermédiaire de la variable `path` du module standard `sys`. Elle est initialisée à l'aide des chemins standard de la version de Python utilisée, enrichis de la liste de chemins que Python a pu trouver dans la variable d'environnement `PYTHONPATH`.

```
>>> import sys
>>> sys.path
['',
 '/home/bob/miniconda3/lib/python36.zip',
 '/home/bob/miniconda3/lib/python3.6',
 '/home/bob/miniconda3/lib/python3.6/lib-dynload',
 '/home/bob/miniconda3/lib/python3.6/site-packages',
 '/home/bob/miniconda3/lib/python3.6/site-packages/Sphinx-1.5.4-py3.6.egg',
 '/home/bob/miniconda3/lib/python3.6/site-packages/setuptools-27.2.0-py3.6.egg',
 '/home/bob/miniconda3/lib/python3.6/site-packages/IPython/extensions',
 '/home/bob/.ipython']
```

La modification de `PYTHONPATH` avant de lancer Python, dépend du shell utilisé.

Par exemple, pour les shells de la famille `sh` :

```
export PYTHONPATH=/home/bob/mydevdir/:$PYTHONPATH
```

Si besoin, on placera ces lignes dans un script shell dédié, ou encore dans le script shell lancé au démarrage de la session afin qu'elles soient exécutées automatiquement.

Sous Windows [1] on pourra utiliser :

```
SET PYTHONPATH=C:\\Users\\Moi\\mydevdir\\;%PYTHONPATH%
```

Si besoin, on pourra aussi positionner les variables d'environnement de façon pérenne *via* le dialogue Windows dédié.

Dans un module, on peut modifier dynamiquement `sys.path` pour que Python aille chercher des modules dans d'autres répertoires.

Attention

Masquage de noms de modules. Les modules sont recherchés dans l'ordre des chemins du `sys.path`. Il est tout à fait possible, volontairement ou non, de masquer un module standard par un module personnel de même nom listé avant.

7.1.3 Emplois et chargements des modules

Un module outil et son utilisation

Soit le module de filtrage de valeurs défini dans le fichier `filtrage.py` :

1. https://ss64.com/nt/set.html

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
# Fichier: filtrage.py
# Limites par défaut pour les filtrages.
FMINI = 100
FMAXI = 500

# On compte le nombre total de valeurs modifiées.
cpt_filtrages = 0
cpt_ajuste = 0

def filtre_serie(lst, mini=FMINI, maxi=FMAXI):
    """Limitation des valeurs d'une liste entre deux limites.

    Construit et retourne une nouvelle liste avec les valeurs
    filtrées. Les valeurs hors limites sont ramenées aux seuils
    mini/maxi indiqués.
    """
    global cpt_filtrages, cpt_ajuste
    res = []
    for v in lst:
        if v < mini:
            res.append(mini)
            cpt_ajuste += 1
        elif v > maxi:
            res.append(maxi)
            cpt_ajuste += 1
        else:
            res.append(v)
        cpt_filtrages += 1
    return res
```

Ce module définit non seulement une fonction `filtre_serie()`, mais aussi deux variables globales `cpt_filtrages` et `cpt_ajuste` et deux constantes `FMINI` et `FMAXI`. Il s'agit d'un script Python comme on en a déjà vu, sauf que si on l'exécute il ne se « passe rien », du moins rien de visible. Le module est bien chargé en mémoire et, si on regarde le *Workspace* de `Pyzo` après l'exécution de ce module, on peut voir que les constantes, les variables globales ainsi que la fonction ont été définies et sont disponibles au niveau de l'espace de noms du module. Il n'y a plus qu'à utiliser cet espace de noms en l'important dans un autre module :

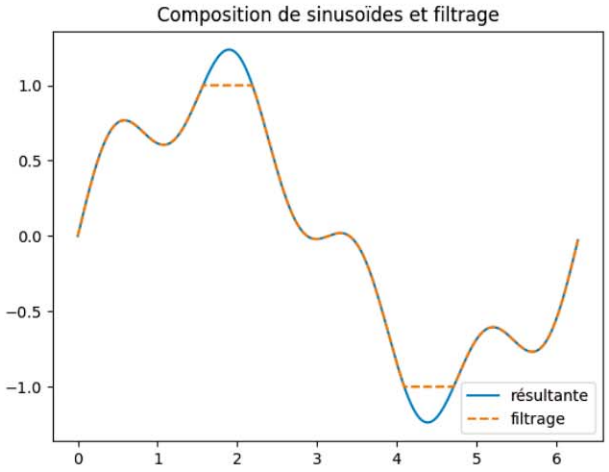
```
# Fichier: sinusoides.py
from math import sin
from matplotlib import pyplot as plt

import filtrage

# On crée des listes de valeurs
liste_x = [x/100 for x in range(628)]
liste_y = [sin(x) for x in liste_x]
liste_y2 = [0.3*sin(4*x) for x in liste_x]
liste_res = [ a+b for a,b in zip(liste_y, liste_y2)]

# On filtre:
liste_res2 = filtrage.filtre_serie(liste_res, -1, 1)
print("Ajusté", filtrage.cpt_ajuste, "valeurs sur", filtrage.cpt_filtrages)

# On …trace
```

Copyright © 2017 Dunod.

```
fig, ax = plt.subplots()
line1 = ax.plot(liste_x, liste_res, label="résultante")
line2 = ax.plot(liste_x, liste_res2, label="filtrage", linestyle='dashed')
ax.legend(loc="lower right")
plt.title("Composition de sinusoïdes et filtrage")
plt.show()
```

On a simplement importé le module `filtrage` par son nom (sans le `py`), ce qui nous a donné accès aux éléments définis dans l'espace de noms correspondant que l'on a pu utiliser.

À l'exécution, on a l'ouverture d'une fenêtre de `matplotlib` pour le tracé des courbes et l'affichage du nombre de valeurs filtrées/corrigées :




```
Ajusté 123 valeurs sur 628
```

Ordre de chargement des modules et du module principal

Le module principal pour Python est le script chargé en premier par l'interpréteur : celui dont on a demandé l'exécution en script principal dans **Pyzo**, ou encore celui qui a été fourni comme argument en ligne de commande à l'interpréteur Python [1].

Pour montrer le chargement des modules, l'exécution de leur code d'initialisation et la définition de la variable globale réservée `__name__` spécifique à chaque module (qui permet d'identifier le module principal), nous allons exécuter les trois scripts suivants :

```
# Fichier mod_A.py
print("Chargement de mod_A")
print("Dans mod_A, __name__ est:", __name__)
print("Fin chargement de mod_A")
```

```
# Fichier mod_B.py
print("Chargement de mod_B")
print("Dans mod_B, __name__ est:", __name__)
print("Import de mod_A dans mod_B")
```

1. Si vous utilisez la ligne de commande pour spécifier le module principal, n'oubliez pas l'extension `.py`, elle est nécessaire.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
import mod_A
print("Fin import de mod_A dans mod_B")
print("Fin chargement de mod_B")
```

```
# Fichier mod_C.py
print("Chargement de mod_C")
print("Dans mod_C, __name__ est:", __name__)
print("Import de mod_A dans mod_C")
import mod_A
print("Fin import de mod_A dans mod_C")
print("Import de mod_B dans mod_C")
import mod_B
print("Fin import de mod_B dans mod_C")
print("Fin chargement de mod_C")
```

L'exécution directe du fichier script `mod_A.py` dans `Pyzo` (`Ctrl+Shift+E`) donne :

```
>>> (executing file "mod_A.py")
Chargement de mod_A
Dans mod_A, __name__ est: __main__
Fin chargement de mod_A
```

On peut voir que la variable globale `__name__` dans le module vaut la chaîne `"__main__"`. Cela indique que `mod_A` est le module principal chargé en premier par l'interpréteur Python.

Exécutons maintenant le fichier script `mod_B.py` dans `Pyzo` (passer explicitement par `Démarrer le script (Ctrl+Shift+E)`, ce qui réinitialise le shell Python pour lancer l'exécution, contrairement à une exécution par le raccourci F5) :

```
>>> (executing file "mod_B.py")
Chargement de mod_B
Dans mod_B, __name__ est: __main__
Import de mod_A dans mod_B
Chargement de mod_A
Dans mod_A, __name__ est: mod_A
Fin chargement de mod_A
Fin import de mod_A dans mod_B
Fin chargement de mod_B
```

On peut voir que, `mod_B` étant maintenant le module principal, sa variable globale `__name__` est définie à `"__main__"` mais que, par contre, la variable globale `__name__` dans `mod_A` est maintenant définie à `"mod_A"` ; cela sera le cas à chaque fois que `mod_A` sera chargé *via* un import dans un autre module et non comme module principal.

Et finalement, exécutons le fichier script `mod_C.py` dans `Pyzo` (toujours en utilisant `Ctrl+Shift+E`) :

```
>>> (executing file "mod_C.py")
Chargement de mod_C
Dans mod_C, __name__ est: __main__
Import de mod_A dans mod_C
Chargement de mod_A
```

Copyright © 2017 Dunod.

```
Dans mod_A, __name__ est: mod_A
Fin chargement de mod_A
Fin import de mod_A dans mod_C
Import de mod_B dans mod_C
Chargement de mod_B
Dans mod_B, __name__ est: mod_B
Import de mod_A dans mod_B
Fin import de mod_A dans mod_B
Fin chargement de mod_B
Fin import de mod_B dans mod_C
Fin chargement de mod_C
```

On vérifie bien que le seul module dont la variable globale `__name__` est `"__main__"` est le module principal chargé en premier par l'interpréteur, `mod_C.py`.

On vérifie aussi que, si le premier import de `mod_A` fait par `mod_C` a réalisé l'initialisation de `mod_A`, le second import de `mod_A` *via* l'import de `mod_B` par `mod_C` n'a pas fait réexécuter le code de `mod_A` : celui-ci n'est exécuté qu'au chargement du module. Une fois un module chargé en mémoire et initialisé, tout nouvel import se limite à aller rechercher son espace de noms.

Notion d'« auto-test »

La valeur de la variable `__name__` nous permet d'identifier le module principal. À partir de là, il est possible de placer du code conditionnel à l'initialisation d'un module qui ne sera exécuté que si celui-ci est le module principal.

On utilise ce mécanisme pour insérer un code d'auto-test du module à la fin de celui-ci, conditionné par `if __name__ == "__main__":`. Le module a donc la structure suivante :

- en-tête ;
- définition des globales / constantes ;
- définition des fonctions et/ou classes ;
- code conditionnel d'auto-test.

```
# fichier cuble_m.py
def cube(x):
    """retourne le cube  de <x>."""
    return x**3

# auto-test ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
if __name__ == "__main__":  # vrai car on est dans le module principal (cube_m.py)
    if cube(9) == 729:
        print("OK !")
    else:
        print("KO !")
```

Utilisation de ce module dans un autre (par exemple celui qui contient le programme principal) :

```
# fichier calcul_cube.py
import cube_m
# On est dans le fichier qui utilise (qui importe) le fichier cube_m.py

# programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
for i in range(1, 5):
    print("cube de", i, "=", cube_m.cube(i))
```

On obtient l'affichage :

```
cube de 1 = 1
cube de 2 = 8
cube de 3 = 27
cube de 4 = 64
```

Autre exemple de codage d'un auto-test dans un module :

```
# fichier validation_m.py
def ok(message) :
    """Retourne True si on saisit <Entrée>, <O>, <o>, <Y> ou <y>,
    False dans tous les autres cas."""
    s = input(message + " (O/n) ? ")
    return True  if s == "" or s[0] in "OoYy" else False

# auto-test ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
if __name__ == '__main__' :
    while True:
        if ok("Encore"):
            print("Je continue")
        else:
            print("Je m'arrête")
            break
```

Exemple d'utilisation :

```
Encore (O/n)?
Je continue
Encore (O/n)? o
Je continue
Encore (O/n)? n
Je m'arrête
```

Module outils utilisable en ligne de commande

De la même façon, il arrive souvent que l'on définisse des modules outils dont on voudrait pouvoir utiliser directement les fonctionnalités en ligne de commande sans avoir besoin d'écrire un second module pour y parvenir.

L'utilisation du mécanisme d'identification du module principal permet facilement cela. Voyons un exemple simple d'affichage de la somme d'une série de valeurs :

```
# fichier outils.py
def aff_somme(*args):
    print("La somme est:", sum(args))

if __name__ == '__main__':
    import sys
    valsnum = [float(x) for x in sys.argv[1:]]
    aff_somme(*valsnum)
```

Si le module est importé normalement, il définit et rend accessible sa fonction `aff_somme()` sans perturber le module qui l'a importé.

Copyright © 2017 Dunod.

```
>>> import outil

>>> outil.aff_somme(1, 8, 2, 9, 5)
La somme est: 25
```

Si le module est utilisé comme module principal en ligne de commande, alors le code principal est activé et l'affichage se fait à partir des valeurs des arguments en ligne de commande. [cf. p. 77]

```
user@host:~$ python3 outil.py 7 4 3 6
La somme est: 20.0
```

7.2 *Batteries included*

On dit souvent que Python est livré « avec les piles » (*batteries included*) tant sa bibliothèque standard, riche de plus de 200 packages et modules, répond aux problèmes courants les plus variés.

Ce survol présente quelques fonctionnalités utiles.

La gestion des chaînes

Le module `string` fournit des constantes comme `ascii_lowercase`, `digits`, `punctuation`... ainsi que la classe `Formatter`, qui peut être spécialisée en sous-classes de *formateurs* de chaînes.

Le module `textwrap` est utilisé pour formater un texte complet : longueur de chaque ligne, contrôle de l'indentation.

Le module `struct` permet de convertir des nombres, des booléens et des chaînes en leur représentation binaire afin de communiquer avec des bibliothèques de bas niveau (souvent en C).

Le module `difflib` permet la comparaison de séquences et fournit des sorties au format standard « diff » ou en HTML.

Enfin, on ne saurait oublier le module `re`, qui offre à Python la puissance des expressions régulières (cf. annexe D p. 181).

La gestion de la ligne de commande

Pour gérer la ligne de commande, Python propose, *via* la liste de chaînes de caractères `sys.argv`, un accès aux arguments fournis au programme par la ligne de commande : `argv[1]`, `argv[2]`... sachant que `argv[0]` est le nom du script lui-même.

Par ailleurs, Python propose un module de *parsing* (analyse) de la ligne de commande, le module `argparse`, qui permet de spécifier les arguments possibles du programme et d'utiliser cette spécification pour analyser la ligne de commande.

C'est un module objet qui s'utilise en trois étapes :

1. Création d'un objet `parser`.
2. Ajout des arguments possibles en utilisant la méthode `add_argument()`. Chaque argument peut déclencher une action particulière spécifiée dans la méthode.
3. Analyse de la ligne de commande par la méthode `parse_args()`.

Enfin, selon les paramètres détectés par l'analyse, on effectue les actions adaptées.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Dans l'exemple suivant, extrait de la documentation officielle du module, on se propose de donner en argument à la ligne de commande une liste d'entiers. Par défaut, le programme retourne le plus grand entier de la liste mais, s'il détecte l'argument `--som`, il retourne la somme des entiers de la liste. De plus, lancé avec l'option `-h` ou `--help`, le module `argparse` fournit automatiquement une documentation du programme :

```
import argparse

# 1. création du parser
parser = argparse.ArgumentParser(description="Gestion d'entiers.")

# 2. ajout des arguments
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help="l'accumulateur (entier)")

parser.add_argument("--som", dest="accumulate", action="store_const",
                    const=sum, default=max,
                    help="somme les entiers (par défaut: donne le maximum)")

# 3. analyse de la ligne de commande
args = parser.parse_args()

# traitement
print(args.accumulate(args.integers))
```

Voici les sorties correspondant aux différents cas de la ligne de commande :

```
$ python argparse.py -h
usage: argparse.py [-h] [--som] N [N ...]

Gestion d'entiers.

positional arguments:
  N           l'accumulateur (entier)

optional arguments:
  -h, --help  show this help message and exit
  --som       somme les entiers (par défaut: donne le maximum)

$ python argparse.py --help
usage: argparse.py [-h] [--som] N [N ...]

Gestion d'entiers.

positional arguments:
  N           l'accumulateur (entier)

optional arguments:
  -h, --help  show this help message and exit
  --som       somme les entiers (par défaut: donne le maximum)

$ python argparse.py 1 2 3 4 5 6 7 8 9
9
```

Copyright © 2017 Dunod.

```
$ python argparse.py --som 1 2 3 4 5 6 7 8 9
45
```

La gestion du temps et des dates

Les modules `calendar`, `time` et `datetime` fournissent les fonctions courantes de gestion du temps [1] et des durées :

```
>>> import calendar, datetime, time
>>> time.asctime(time.gmtime(0))   # l'origine des temps Unix
'Thu Jan  1 00:00:00 1970'
>>> moon_apollo11 = datetime.datetime(1969, 7, 20, 20, 17, 40)
>>> vendredi_precedent, un_jour = moon_apollo11, datetime.timedelta(days=1)
>>> while vendredi_precedent.weekday() != calendar.FRIDAY:
...     vendredi_precedent -= un_jour
>>> vendredi_precedent.strftime("%A, %d-%b-%Y")
'Friday, 18-Jul-1969'
```

Algorithmes et types de données `collection`

Le module `bisect` fournit des fonctions de recherche rapide dans des séquences triées. Le module `array` propose un type semblable à la liste, mais plus rapide et plus efficace au niveau du stockage, car de contenu homogène (assimilables aux « tableaux » dans de nombreux langages).

Le module `heapq` [2] gère des listes organisées en file d'attente dans lesquelles les manipulations des éléments assurent que la file reste toujours organisée en arbre binaire (structure de données permettant des ajouts en maintenant l'ordre des données).

Python propose, *via* le module `collections`, la notion de type tuple nommé avec le type `namedtuple` (il est bien sûr possible d'avoir des tuples nommés emboîtés). En plus de l'accès par index, ceux-ci permettent d'accéder aux valeurs du tuple par des noms, donnant un sens aux valeurs manipulées :

```
>>> import collections
>>> import math
>>> Point = collections.namedtuple('Point', 'x y z')  # description du type
>>> p1 = Point(1.2, 2.3, 3.4)  # on instancie deux 'Point'
>>> p2 = Point(-0.6, 1.4, 2.5)
>>> d = math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2 + (p1.z - p2.z)**2)
>>> print("Distance :", d)
Distance : 2.20454076850486
```

Dans le même module `collections`, le type `defaultdict` utilise une fonction à appeler pour produire une valeur par défaut lorsqu'on utilise une clé qui n'est pas encore dans le dictionnaire. La valeur produite est stockée dans le dictionnaire et retournée comme si elle avait déjà été présente. Dans les exemples ci-après, nous utilisons simplement les types de base Python pour générer des valeurs par défaut (`[]` puis `0`) utilisées ensuite dans des expressions :

```
>>> from collections import defaultdict
>>> s = [('y', 1), ('b', 2), ('y', 3), ('b', 4), ('r', 1)]
```

1. La gestion du temps sur une période historique peut être particulièrement ardue en raison des multiples déclinaisons des bases de datation et des corrections qui y ont été apportées. Pour les usages avancés, il est conseillé de piocher dans les modules tiers disponibles sur le *Python Package Index*, comme `convertdate` ou `jdcal`.

2. On utilise aussi en informatique le terme « tas ».

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
>>> d = defaultdict(list)  # list() produira une nouvelle liste vide []
>>> for k, v in s:
...     d[k].append(v)
>>> d.items()
dict_items([('y', [1, 3]), ('b', [2, 4]), ('r', [1])])
>>> s = 'mississippi'
>>> d = defaultdict(int)  # int() produira un entier nul 0
>>> for k in s:
...     d[k] += 1
>>> d.items()
dict_items([('m', 1), ('i', 4), ('s', 4), ('p', 2)])
```

Et tant d'autres domaines...

Beaucoup d'autres sujets pourraient être explorés :

— accès au système ;
— utilitaires fichiers ;
— programmation réseau ;
— persistance ;
— fichiers XML ;
— compression ;
— ...

7.3 Python scientifique

Depuis sa création, Python permet l'écriture de modules en langage C qui sont ensuite compilés sous forme de librairies dynamiques.

Le mode d'exécution des modules écrits en Python, avec la compilation du source en *bytecode* puis l'interprétation de ce *bytecode* dans une machine virtuelle [cf. p. 12], s'il apporte une grande souplesse, reste très en deçà au niveau des performances par rapport à du code C compilé pour une exécution directe par le processeur.

La capacité de Python à permettre de façon transparente [a] le chargement de modules compilés donne accès dans de nombreux domaines à des librairies très efficaces.

a. Lorsqu'on importe un module en Python, on ne sait pas *a priori* s'il est écrit en C et compilé ou bien écrit en Python – son utilisation est identique.

Dans les années 1990, Travis Oliphant et d'autres commencèrent à élaborer des outils efficaces de traitement des données numériques : `Numeric`, `Numarray`, et enfin `NumPy` en 2005.

`SciPy`, bibliothèque d'algorithmes scientifiques, a également été créée à partir de ces outils numériques. Au début des années 2000, John Hunter crée `matplotlib`, un module de tracé de graphiques 2D. À la même époque, Fernando Perez crée `IPython` en vue d'améliorer l'interactivité et la productivité en Python scientifique, outil qui devait évoluer jusqu'à l'actuel `Jupyter Notebook`.

En moins de dix ans, les outils essentiels pour faire de Python un langage scientifique performant étaient en place.

Copyright © 2017 Dunod.

7.3.1 Bibliothèques mathématiques et types numériques

On rappelle que Python offre la bibliothèque `math`, qui fournit les fonctions de base pour les calculs trigonométriques, logarithmiques, d'arrondis... ainsi que diverses constantes usuelles. La bibliothèque `cmath` fournit ces mêmes fonctions, mais avec le support des nombres complexes.

```
>>> import math
>>> math.pi / math.e
1.1557273497909217
>>> math.exp(1e-5) - 1
1.0000050000069649e-05
>>> math.log(10)
2.302585092994046
>>> math.log(1024, 2)
10.0
>>> math.cos(math.pi/4)
0.7071067811865476
>>> math.atan(4.1/9.02)
0.4266274931268761
>>> math.hypot(3, 4)
5.0
>>> math.degrees(1)
57.29577951308232
```

Par ailleurs, Python propose en standard les modules `fraction` et `decimal` pour offrir un support à ces types de données spécifiques (le type décimal est entre autres utilisé en comptabilité pour ne pas avoir les effets d'arrondi non contrôlé sur la représentation des nombres flottants lors des calculs). Ces types sont utilisables normalement avec les types entier et flottant standard :

```
from fractions import Fraction
from decimal import Decimal, getcontext

f1 = Fraction(16, -10)      # -8/5
f2 = Fraction(123)          # 123
f3 = Fraction(' -3/10 ')    # -3/10
f4 = Fraction('-.125')      # -1/8
f5 = Fraction('7e-6')       # 7/1000000
f6 = f1 + f3                # -19/10
f7 = f4 * 512               # -64/1

d1 = Decimal(1)
d2 = Decimal(7)
getcontext().prec = 3
d3 = d1 / d2                # 0.143
getcontext().prec = 6
d4 = d1 / d2                # 0.142857
getcontext().prec = 18
d5 = d1 / d2                # 0.142857142857142857
d6 = (d1 /d2) * 100         # 14.2857142857142857
```

Enfin la bibliothèque standard `random` propose plusieurs fonctions de nombres aléatoires ou permettant des opérations aléatoires sur les conteneurs séquences. Elle fournit différents algorithmes de génération de nombres aléatoires avec différentes répartitions statistiques.

Depuis Python 3.4, la bibliothèque standard `statistics` fournit les fonctions de base pour les calculs statistiques courants.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

7.3.2 L'interpréteur IPython

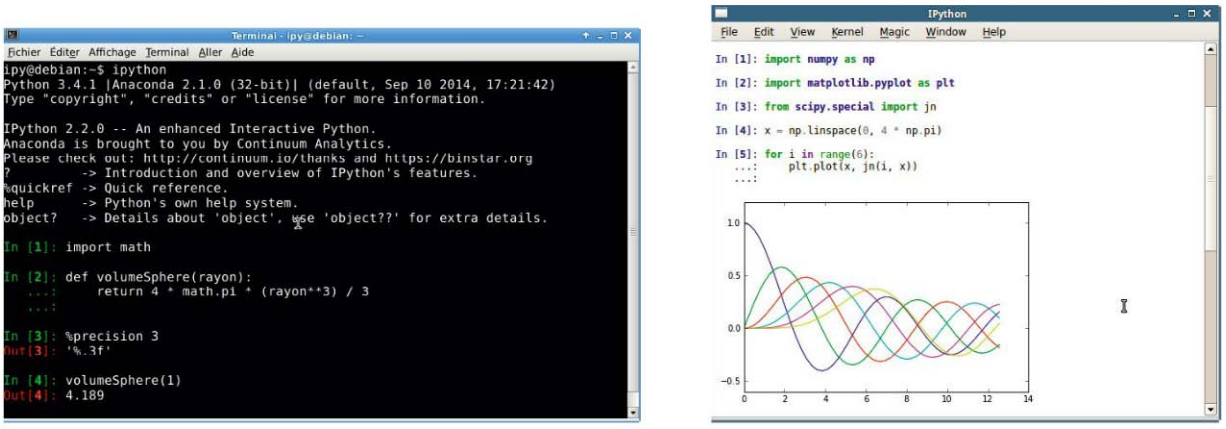
Remarque

✓ On peut dire que `IPython` est devenu *de facto* l'interpréteur standard du Python scientifique.

En mars 2013, ce projet a valu le prestigieux prix du développement logiciel libre par la Free Software Foundation à son créateur Fernando Perez.

`IPython` [1] est disponible en trois déclinaisons (☞ Fig. 7.1 et ☞ Fig. 7.2) :

- **`ipython`**, l'interpréteur de base ;
- **`ipython qtconsole`**, sa version améliorée dans une fenêtre graphique de la bibliothèque graphique Qt, agrémentée d'un menu ;
- **`Jupyter Notebook`**, sa dernière variante qui offre une interface simple mais très puissante dans le navigateur web par défaut. Son utilisation est décrite en détail dans les exercices en ligne.



(a) `ipython` (b) `ipython qtconsole`

FIGURE 7.1 – `IPython` en mode console texte ou graphique

La version *Jupyter Notebook* mérite une mention spéciale : chaque cellule du notebook peut être du code, des figures, du texte enrichi (y compris des formules mathématiques), des vidéos, etc.

La figure 7.2 présente un exemple de tracé interactif.

Objectifs

D'après ses concepteurs, les objectifs d'`IPython` sont les suivants :

- fournir un **interpréteur Python plus puissant** que celui par défaut. `IPython` propose de nombreuses caractéristiques comme l'*introspection d'objet*, l'*accès au shell système* ainsi que ses propres commandes permettant une grande interaction avec l'utilisateur ;
- proposer un **interpréteur embarquable** et prêt à l'emploi pour vos programmes Python. `IPython` s'efforce d'être un environnement efficace à la fois pour le développement de code Python et pour la résolution des problèmes liés à l'utilisation d'objets Python (cf. les caractéristiques qui suivent) ;

1. On distingue le nom de l'outil « IPython » du nom de la commande `ipython`, commande qui permet d'invoquer l'interpréteur depuis une console.

Copyright © 2017 Dunod.

— permettre le **test interactif des bibliothèques graphiques**, entre autres `tkinter`, `wxPython`, `PyGTK`, `PyQt` (alors que `IDLE` ne le permet qu'avec des applications `tkinter`).

Quelques caractéristiques

— `IPython` est auto-documenté.
— Il offre la coloration syntaxique.
— Les *docstrings* des objets Python sont disponibles en accolant un « ? » au nom de l'objet ou « ?? » pour une aide plus détaillée.
— Il numérote les entrées et les sorties pour permettre de s'y référer.
— Il organise les sorties : messages d'erreur ou retour à la ligne entre chaque élément d'une liste si on l'affiche.
— Il offre l'auto-complétion avec la touche `TAB` :
  — l'auto-complétion trouve les variables qui ont été déclarées,
  — elle trouve les mots clés et les fonctions locales,
  — la complétion des méthodes sur les variables tient compte du type actuel de cette dernière,
  — par contre la complétion ne tient pas compte du contexte.
— Il propose un historique persistant (même si on quitte l'interpréteur, on peut retrouver les dernières commandes par l'historique) :
  — recherche dans l'historique avec les flèches du clavier,
  — isole dans l'historique les entrées multilignes,
  — peut rappeler les entrées et sorties précédentes.
— Il contient des raccourcis et des alias. On peut en afficher la liste en tapant la commande `lsmagic`.
— Il permet d'exécuter des commandes système (shell) en les préfixant par un point d'exclamation. Par exemple `!ls` sous Linux ou OSX, ou `!dir` dans une fenêtre de commande Windows.

7.3.3 La bibliothèque `NumPy`

Introduction

Le module `numpy` est la boîte à outils indispensable pour faire du calcul scientifique avec Python [1].

Pour modéliser les vecteurs, matrices et, plus généralement, les tableaux à n dimensions, `numpy` fournit le type `ndarray`.

On note des différences majeures entre les tableaux `numpy` et les listes (resp. les listes de listes) qui pourraient nous servir à représenter des vecteurs (resp. des matrices) :

— les **tableaux sont homogènes**, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc. ;
— la **taille des tableaux est fixée** à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme on le ferait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ces contraintes sont en fait des avantages pour le calcul numérique :

1. Cette introduction est reprise de l'excellent mémento de Jean-Michel Ferrard, avec son aimable autorisation http://mathprepa.fr/python-project-euler-mpsi-mp/.

© Dunod. Toute reproduction non autorisée est un délit.
Copyright © 2017 Dunod.

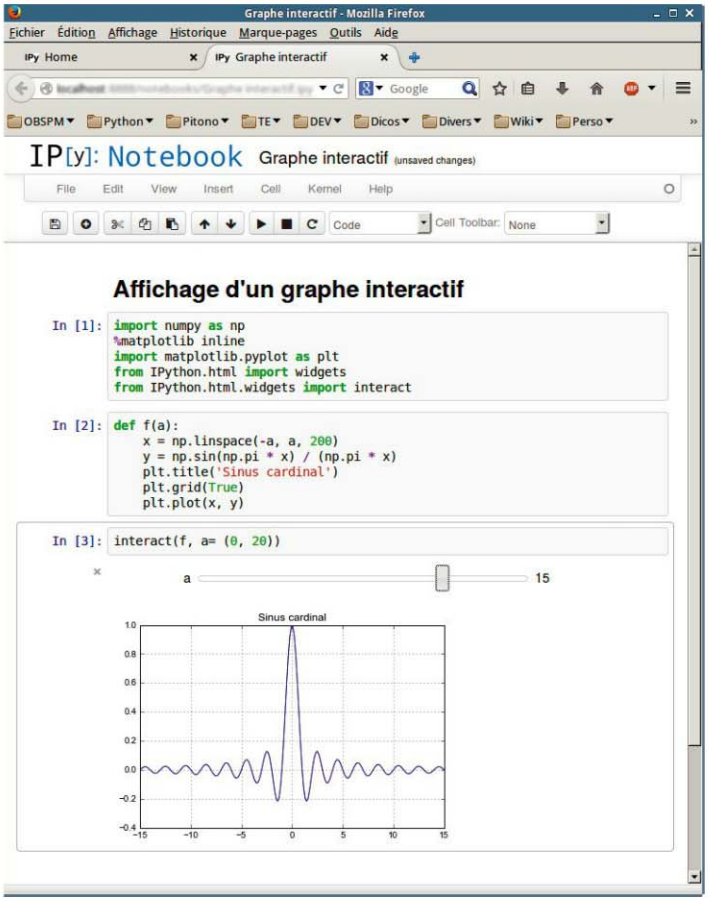


FIGURE 7.2 – `Jupyter Notebook`

— le format d'un tableau `numpy` et la taille des objets qui le composent étant fixés, **l'empreinte du tableau en mémoire est invariable** ;
— les **opérations sur les tableaux sont optimisées** en fonction du type des éléments et sont beaucoup plus rapides qu'elles ne le seraient sur des listes équivalentes.

Exemples

Dans ce premier exemple, on définit un tableau `a` d'entiers puis on le multiplie *globalement*, c'est-à-dire sans utiliser de boucle Python, par le scalaire `2.5`. On définit de même le tableau `d`, qui est affecté en une seule instruction à `a + b`.

```
In [1]: import numpy as np
In [2]: a = np.array([1, 2, 3, 4])
In [3]: a, a.dtype      # a est un tableau d'entiers
Out[3]: (array([1, 2, 3, 4]), dtype('int64'))
In [4]: b = a * 2.5
In [5]: b, b.dtype      # b est bien devenu un tableau de flottants
Out[5]: (array([  2.5,   5. ,   7.5,  10. ]), dtype('float64'))
In [6]: a @ b           # multiplication matricielle (à partir de Python 3.6)
Out[6]: 75.0
In [7]: c = np.array([5, 6, 7, 8])
In [8]: d = b + c
In [9]: d, d.dtype      # transtypage automatique en flottants
```

Copyright © 2017 Dunod.

```
Out[9]: (array([  7.5,  11. ,  14.5,  18. ]), dtype('float64'))
```

L'exemple suivant définit un tableau `positions` de 10 000 000 lignes et 2 colonnes, formant des positions aléatoires. Les vecteurs colonnes `x` et `y` sont extraits du tableau `position`. On affiche le tableau et le vecteur `x` avec 3 chiffres après le point décimal. On calcule (bien sûr *globalement*) le vecteur des distances euclidiennes à un point particulier (x_0, y_0) et on affiche l'indice du tableau de la distance minimale à ce point.

```
In [1]: import numpy as np
In [2]: positions = np.random.rand(10000000, 2)
In [3]: x, y = positions[:, 0], positions[:, 1]
In [4]: %precision 3
Out[4]: '%.3f'
In [5]: positions
Out[5]:
array([[ 0.861,  0.373],
       [ 0.627,  0.935],
       [ 0.224,  0.058],
       ...,
       [ 0.628,  0.66 ],
       [ 0.546,  0.416],
       [ 0.396,  0.625]])
In [6]: x
Out[6]: array([ 0.861,  0.627,  0.224, ...,  0.628,  0.546,  0.396])
In [7]: x0, y0 = 0.5, 0.5
In [8]: distances = (x - x0)**2 + (y - y0)**2
In [9]: distances.argmin()
Out[9]: 4006531
```

Ce type de traitement très efficace et élégant est typique des logiciels analogues comme MATLAB.

7.3.4 La bibliothèque `matplotlib`

Cette bibliothèque que nous avons déjà utilisée (cf. § 1.1.1 p. 2 et cf. § 7.1.3 p. 73) permet toutes sortes de représentations [1] de graphes 2D (☞ Fig. 7.3) et quelques-unes en 3D :

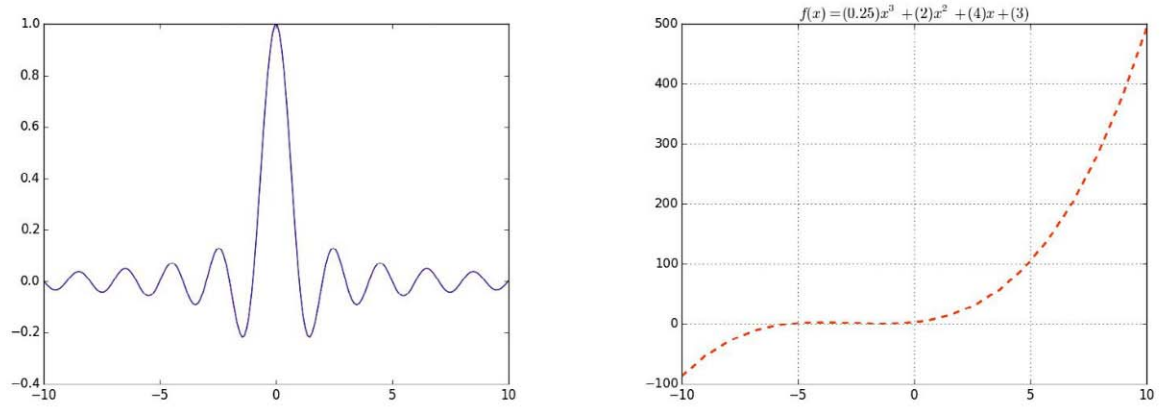
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 200)   # 200 valeurs flottantes réparties entre -10 et 10
y = np.sin(np.pi * x)/(np.pi * x)

plt.plot(x, y)
plt.show()
```

Ce second exemple améliore le tracé. Il utilise le *style objet* de `matplotlib` :

```
import numpy as np
import matplotlib.pyplot as plt

def plt_arrays(x, y, title='', color='red', linestyle='dashed', linewidth=2):
```

1. Notons que sa syntaxe de base a été pensée pour ne pas dépayser l'utilisateur de la bibliothèque graphique de MATLAB, et ainsi lui offrir une alternative gratuite…

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.



(a) Un tracé simple

(b) Un graphe décoré

FIGURE 7.3 – Exemples de tracé avec `matplotlib`

```
    """Définition des caractéristiques et affichage d'un tracé y(x)."""
    fig = plt.figure()
    axes = fig.add_subplot(111)
    axes.plot(x, y, color=color, linestyle=linestyle, linewidth=linewidth)
    axes.set_title(title)
    axes.grid()
    plt.show()


def f(a, b, c, d):
    x = np.linspace(-10, 10, 20)
    y = a*(x**3) + b*(x**2) + c*x + d
    title = '$f(x) = (%s)x^3 + (%s)x^2 + (%s)x + (%s)$' % (a, b, c, d)
    plt_arrays(x, y, title=title)


f(0.25, 2, 4, 3)
```

7.3.5 La bibliothèque SymPy

`SymPy` est une bibliothèque en pur Python spécialisée dans le calcul formel à l'instar de Mapple ou Mathematica [1]. Elle permet de faire du calcul arithmétique formel (basique), de l'algèbre, des mathématiques différentielles, de la physique, de la mécanique...

Il est facile de se familiariser avec `SymPy` (☞ Fig. 7.4) en l'expérimentant directement sur Internet [2]. On peut, bien sûr, l'utiliser localement après l'avoir installée par `conda` [cf. p. 171].

1. On parle aussi de CAS (*Computer Algebra System*).
2. `http://live.sympy.org` (☞ Fig. 7.4). Attention, l'interface utilise Python 2.7.5

Copyright © 2017 Dunod.

7.4 Bibliothèques tierces

7.4.1 Une grande diversité

Outre les nombreux modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;
- 3D ;
- ...

Le site pypi.python.org/pypi [1] recense et donne accès à des milliers de modules et de packages !

Ceux-ci sont facilement installables *via* conda [cf. p. 171] ou *via* l'outil en ligne de commande `pip3`.

```
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Warning: this shell runs with SymPy 1.0 and so examples pulled from
other documentation may provide unexpected results.
Documentation can be found at http://docs.sympy.org/1.0.

>>> expr = (x + y)**5
>>> expand(expr)
```

$$x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$$

```
>>> sin(x).series(x, 0, 5)
```

$$x - \frac{x^3}{6} + \mathcal{O}\left(x^5\right)$$

FIGURE 7.4 – Expérimenter `SymPy` avec « live sympy »

7.4.2 Un exemple : la bibliothèque `Unum`

Cette bibliothèque permet de calculer en tenant compte des unités du système SI (système international d'unités).

Voici un exemple de session interactive :

```
>>> from unum.units import *
>>> distance = 100*m
>>> temps = 9.683*s
>>> vitesse = distance / temps
>>> vitesse
10.327377878756584 [m/s]
>>> vitesse.asUnit(mile/h)
```

1. The python package index

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
23.1017437978 [mile/h]
>>> acceleration = vitesse/temps
>>> acceleration
1.0665473385063085 [m/s2]
```

7.5 Packages

Outre le module, un deuxième niveau d'organisation permet de structurer le code : les fichiers Python peuvent être organisés en une **arborescence de répertoires** appelée *paquet*, en anglais *package*.

Définition

✍ Un **package** est un module contenant d'autres modules. Les modules d'un package peuvent être des *sous-packages*, ce qui donne une structure arborescente.

Pour être reconnu comme un package valide, chaque répertoire du paquet doit posséder un fichier `__init__`, qui peut soit être vide, soit contenir du code d'initialisation.

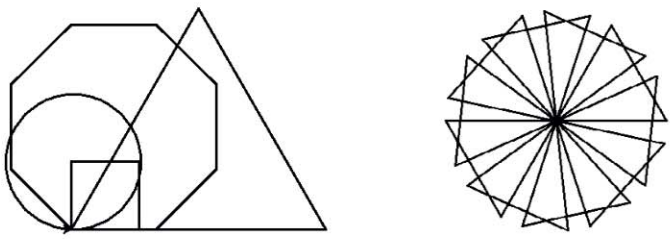
Pour accéder aux modules ou aux sous-packages qui composent un package, on utilise simplement la notation pointée des espaces de noms :

```
>>> import os.path
>>> os.path.expanduser("~/toto")
'/home/bob/toto'
```

7.6 Exercices

1. À l'aide du module `turtle` pour la partie graphique [1], écrivez une fonction `polygone_regulier` qui permet de tracer des polygones réguliers à n cotés (paramétrable) ayant chacun la longueur spécifiée. Ce module comportera un auto-test qui vérifiera le bon fonctionnement de la fonction.

   Utiliser ce module pour écrire une fonction qui permet de tracer n polygones réguliers en démarrant à des angles répartis régulièrement sur un tour complet (☞ Fig. 7.5).



(a) Tests du module (b) Script principal

FIGURE 7.5 – Exemples de tracés de polygones avec `Turtle`

1. Module qui permet de réaliser très facilement des graphiques à l'aide d'un crayon virtuel – à la façon du langage LOGO. Se référer à l'aide-mémoire https://perso.limsi.fr/pointal/python:turtle:accueil.

Copyright © 2017 Dunod.

2. Un tableau contient n entiers ($2 < n < 100$) aléatoires tous compris entre 0 et 500. Vérifier qu'ils sont tous différents.

3. Proposer une version plus simple du problème précédent en comparant les longueurs des tableaux avant et après traitement ; le traitement consiste à utiliser une structure de données contenant des éléments uniques.

4. ✠ Nombres parfaits et nombres chanceux.

   Définitions :
   — on appelle *nombre premier* tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité ;
   — on appelle *diviseur propre* de n, un diviseur quelconque de n, n exclu ;
   — un entier naturel est dit *parfait* s'il est égal à la somme de tous ses diviseurs propres ;
   — un entier `n` tel que : $(n + i + i^2)$ est premier pour tout `i` dans $[0, n-2]$ est dit *chanceux*.

   Écrire un fichier (`parfait_chanceux_m.py`) définissant quatre fonctions : `somDiv`, `estParfait`, `estPremier`, `estChanceux` et un auto-test :

   — la fonction `somDiv()` retourne la somme des diviseurs propres de son argument ;
   — les trois autres fonctions vérifient que leur argument possède la propriété donnée par leur définition et retournent un booléen. Si par exemple la fonction `estPremier()` vérifie que son argument est premier, elle retourne `True`, sinon elle retourne `False`.

   L'auto-test doit comporter quatre appels à la fonction `isclose()` du module `math` permettant de tester `somDiv(12)`, `estParfait(6)`, `estPremier(31)` et `estChanceux(11)`.

   Puis écrire le programme principal (`parfait_chanceux.py`) qui comporte :

   — l'initialisation de deux listes : `parfaits` et `chanceux` ;
   — une boucle de parcours de l'intervalle $[2, 1000]$ incluant les tests nécessaires pour remplir ces listes ;
   — enfin l'affichage de ces listes.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Chapitre 8

La programmation orientée objet

La *programmation orientée objet* (ou POO) :

- permet de mieux modéliser la réalité en concevant des modèles d'objets, les *classes* ;
- ces classes permettent de construire des *objets* interactifs entre eux et avec le monde extérieur ;
- les objets sont créés indépendamment les uns des autres, grâce à l'*encapsulation*, mécanisme qui permet d'embarquer leurs propriétés ;
- les classes permettent d'éviter au maximum l'emploi des variables globales ;
- enfin les classes offrent un moyen économique et puissant de construire de nouveaux objets à partir d'objets préexistants.

8.1 Origine et évolution

La programmation orientée objet, qui a fait ses débuts dans les années 1960 avec des réalisations dans le langage Lisp, a été plus formellement définie avec les langages Simula (fin années 1960) puis SmallTalk (années 1970). Elle n'a depuis cessé de se diffuser dans les différents langages de programmation, faisant évoluer les langages anciens (comme le Fortran, le Cobol ou le C) pour y intégrer ses concepts, et étant même incontournable dans certains langages plus récents (Java).

Les langages de programmation ont quasi tous un moyen de regrouper des données qui doivent aller ensemble. Ce moyen, souvent nommé par les termes anglais *structure* ou *record*, contient des « attributs » ou « champs » [1]. Ces regroupements de données permettent de décrire informatiquement les caractéristiques d'éléments du monde réel, de les modéliser.

1. *Fields* en anglais.

Copyright © 2017 Dunod.

Les langages à programmation objet ajoutent à ces caractéristiques l'association de méthodes de traitement qui permettent de modéliser les interactions qui se produisent entre les éléments du monde réel ainsi que les actions que peuvent réaliser ces éléments. Ainsi sont définis des objets qui encapsulent données et méthodes en des ensembles cohérents pouvant « communiquer » avec d'autres objets (on utilise souvent le terme « envoi de messages » pour les appels de méthodes) pour réaliser au final la tâche désirée.

Les analogies avec le monde réel, et les similitudes entre eux d'objets modélisés, ont conduit à définir des familles d'objets, les classes, et des relations entre ces familles : l'héritage (`X` est-un-genre-de `Y`) et la composition (`X` a-un `Y`).

Cette catégorisation en classes, en relations entre classes, et en échanges entre objets, est un des moyens utilisés pour l'analyse et la modélisation des problèmes en vue de les résoudre informatiquement. Nous verrons dans ce chapitre la représentation de ces éléments *via* des diagrammes de la notation UML [1], notation très répandue qui permet à plusieurs personnes d'échanger sur des analyses en utilisant des bases communes indépendantes du langage de programmation – nous ne rentrerons toutefois pas dans les finesses de spécification que permet cette notation, il y a des ouvrages dédiés à cela.

Remarque

✓ En Python, la programmation orientée objet est partout, mais son utilisation explicite n'est pas forcée. En effet, tout ce qui est manipulé en Python est objet issu d'une famille (classe) et possède des attributs et éventuellement des méthodes. On le pressent avec les listes, les chaînes ou les fichiers et la notation pointée entre une variable et la méthode à lui appliquer. C'est vrai aussi avec les types de base comme les entiers ou les flottants. Mais c'est également vrai avec les éléments de programmation utilisés, par exemple un module Python : une fois qu'il est chargé, c'est un objet de la classe module qui a différents attributs dont sa documentation, son fichier d'origine, son nom, un dictionnaire qui correspond à son espace de noms. De la même façon, une fonction écrite en Python est un objet de la classe `function` avec différents attributs dont son nom, ses paramètres par défaut, son code à exécuter lorsqu'elle est appelée. C'est un des atouts de Python, qui permet l'introspection, présentée au chapitre 10.1.1.

8.2 Terminologie

Le vocabulaire de base de la programmation orientée objet

Une *classe* est une famille d'objets équivalente à un nouveau type de données. On connaît déjà par exemple les classes `list` ou `str` et les nombreuses méthodes permettant de les manipuler en utilisant la notation pointée :

- `lst = [3, 5, 1]`
- `lst.sort()`
- `[4, 1, 7].index(1)`
- `s = "casse"`
- `s.upper()`

Un *objet* ou une *instance* est un exemplaire particulier d'une classe. Par exemple `[3, 5, 1, 9]` et `[4, 1, 7]` sont des instances de la classe `list` et `"casse"` est une instance de la classe `str`. On peut manipuler ces instances à l'aide de leur représentation littérale ou bien *via* des variables qui les référencent.

1. *Unified Modeling Language.*

Copyright © 2017 Dunod.

Les objets ont donc généralement deux sortes d'attributs : les données nommées simplement *attributs* et les fonctions applicables appelées *méthodes*.

Par exemple un objet de la classe `complex` possède :

— deux attributs : `imag` et `real` ;
— plusieurs méthodes, comme `conjugate()`, `abs()`...

La plupart des classes *encapsulent* à la fois les données et les méthodes applicables aux objets. Par exemple un objet `str` contient une chaîne de caractères Unicode (les données) et de nombreuses méthodes.

On peut définir un objet comme une *capsule* contenant des attributs et des méthodes :

objet = attributs + méthodes

Les classes peuvent aussi avoir elles-mêmes des attributs et méthodes, directement liés, existant au niveau de la classe indépendamment des objets. On parle d'attribut de classe et de méthode de classe. Les attributs de classe sont partagés entre tous les objets de cette classe, et les méthodes de classe peuvent travailler sans avoir besoin d'un objet.

8.3 Définition des classes et des instanciations d'objets

8.3.1 L'instruction `class`

Cette instruction composée permet d'introduire la définition d'une nouvelle classe (c'est-à-dire d'un nouveau type de données). Dans cette définition on trouve d'un côté des attributs liés directement à la classe, d'un autre des attributs dont l'existence est liée aux objets de cette classe. Les premiers seront créés directement dans le corps de l'instruction composée, les seconds seront créés dans une méthode (fonction) spéciale d'initialisation des objets.

Les méthodes définies dans la classe sont en général applicables à des objets, et pour cela elles prennent un premier paramètre qui est l'objet manipulé (par convention appelé `self`).

Syntaxe

`class` est une instruction composée. Elle comprend un en-tête (avec *docstring*) et un corps indenté :

```
class MaClasse:
    """Documentation de la classe MaClasse."""
    # définition de la classe : attributs de classe, méthodes
    attcl = 5
    def meth(self):
        pass
```

Dans cet exemple de syntaxe, `MaClasse` est le nom de la classe (utilise conventionnellement des noms en notation CapitalizedWords [cf. p. 17]), qui sera utilisé pour accéder à ses attributs. Les attributs de classe et les méthodes sont définis dans le corps de la classe, ici l'attribut `attcl` et la méthode `meth`.

La nouvelle classe définie est elle-même un objet de la classe `type`, c'est aussi un nouvel espace de noms dans lequel on retrouve les attributs et méthodes qui ont été définis dans son corps :

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

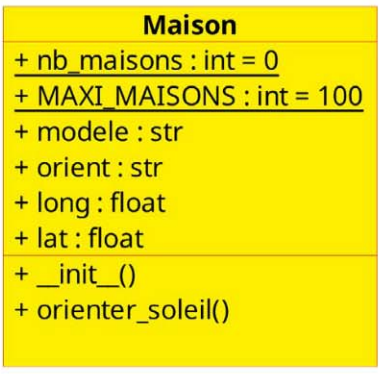
```
>>> MaClasse
<class '__main__.MaClasse'>
>>> type(MaClasse)
<class 'type'>
>>> MaClasse.attcl
5
>>> MaClasse.meth
<function MaClasse.meth at 0x7f49f0088378>
```

8.3.2 L'instanciation et ses attributs

- Les classes sont des **fabriques d'objets**. En utilisant cette métaphore, **on construit d'abord l'usine avant de produire des objets !**
- On **instancie** un objet (c'est-à-dire qu'**on le produit à partir de l'usine**) en appelant sa classe comme s'il s'agissait d'une fonction :
  - l'objet instance de la classe est alors créé « nu » en mémoire ;
  - la méthode spéciale de la classe, `__init__()` est ensuite appelée avec ce nouvel objet en premier paramètre suivi des autres paramètres transmis *via* l'appel à la classe. Cette méthode est chargée d'initialiser le nouvel objet (si cette méthode n'est pas définie, le nouvel objet reste « nu »), nous reviendrons ultérieurement dessus. Chaque nouvel objet créé définit un nouvel espace de noms.

Prenons comme exemple une modélisation de maisons au niveau urbanistique, que l'on va vouloir positionner et orienter pour construire un nouveau quartier.

Si l'on construit un diagramme UML [1] de la classe correspondante, on a le schéma suivant qui liste deux attributs de classe (soulignés), quatre attributs d'instance et une méthode pour la classe Maison (UML permet de préciser les caractéristiques des éléments ; typiquement on a ici les types des données, des valeurs par défaut, et le fait que toutes ces données sont d'accès public avec un `+` devant leur nom) :



Partons de la définition de classe correspondante en Python, et utilisons-la (après une courte explication, nous reviendrons par la suite plus en détail sur les différents aspects des définitions de classe, en utilisant des exemples simplifiés) :

```
class Maison:                       # L'usine
    """Définition d'une maison."""
```

1. Des outils comme ArgoUML, Umbrello (utilisé pour nos exemples), Violet UML Editor, UMLet, Dia... permettent de dessiner facilement ces diagrammes d'analyse.

Copyright © 2017 Dunod.

```
    nb_maisons = 0                  # Attribut de classe
    MAXI_MAISONS = 100              # Attribut constant de classe

    def __init__(self, modele, longitude, latitude, orientation):
        """Initialiseur de Maison."""
        if Maison.nb_maisons >= Maison.MAXI_MAISONS:
            raise RuntimeError("Trop de maisons construites")
        self.modele = modele        # Attribut d'instance
        self.lng = longitude        # Attribut d'instance
        self.lat = latitude         # Attribut d'instance
        self.orient = orientation   # Attribut d'instance
        Maison.nb_maisons += 1      # Modification attribut de classe

    def orienter_soleil(self):
        "Changement du sens d'orientation de la maison vers le sud."
        self.orient = "sud"         # Modification attribut d'instance

m1 = Maison("Cheverny", 2.08333, 48.650002, "nord")
print(m1)
m2 = Maison("Chambord", 2.08313, 48.650042, "est")
print(m2)
print(m1.modele, m2.modele)
print(Maison.nb_maisons)
print(m2.orient)
m2.orienter_soleil()
print(m2.orient)
print(m1.orient)
```

Affiche :

```
<__main__.Maison object at 0x7f1e00672f98>      # l'instance m1
<__main__.Maison object at 0x7f1e00672f28>      # l'instance m2
Cheverny Chambord                               # les modèles de m1 et m2
2           # le nombre de maisons créées
est         # l'orientation initiale de m2
sud         # celle après  orienter_soleil
nord    # l'orientation de m1
```

Dans cet exemple on a créé des variables `m1` et `m2` référençant deux objets construits en appelant la classe `Maison` comme s'il s'agissait d'une fonction. Les arguments passés à la classe ont été retransmis dans la méthode d'initialisation `__init__()`. Celle-ci s'est chargée de stocker ces valeurs reçues en arguments comme attributs de l'objet « self » (respectivement `m1` puis `m2`). Elle a par ailleurs mis à jour l'attribut de classe `nb_maisons`, qui compte le nombre de maisons créées.

On peut voir les accès aux attributs de `m1` et de `m2` dans leurs espaces de noms respectifs, et aussi à l'attribut de classe `nb_maisons` de `Maison`.

Enfin un appel à la méthode `orienter_soleil()` sur `m2`, avec la notation pointée, a bien modifié l'attribut de cet objet, sans modifier celui de l'autre objet (chaque objet définit un espace de noms qui lui est propre).

Définition

✍ Une variable définie au niveau d'une classe (comme `nb_maisons` dans la classe `Maison`) est appelée **attribut de classe** et est partagée par tous les objets instances de cette classe.

Une variable définie au niveau d'un objet (comme `orient` dans les objets `m1` et `m2`) est appelée **attribut d'instance** et est liée uniquement à l'objet pour lequel elle est définie.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

8.3.3 Retour sur les espaces de noms

On a déjà vu les espaces de noms [1] locaux (lors de l'appel d'une fonction), globaux (liés aux modules) et internes (fonctions standard), ainsi que la règle LGI (« Local Global Interne », cf. section 6.2, p. 66), qui définit dans quel ordre ces espaces sont parcourus pour résoudre un nom.

Les classes ainsi que les objets instances définissant de nouveaux espaces de noms, il y a là aussi des règles pour résoudre les noms entre ces espaces.

Définition

- Un **nom non qualifié** est un nom accessible directement sans notation pointée devant lui.
- Un **nom qualifié par une classe** est un nom précédé par une notation pointée spécifiant une classe.
- Un **nom qualifié par une instance** est un nom précédé par une notation pointée spécifiant une variable d'instance (le paramètre `self` passé aux méthodes étant bien une variable référençant l'instance manipulée dans la méthode).

Pour les accès aux attributs en lecture

- **Noms non qualifiés** : ils sont recherchés simplement suivant la règle LGI.
- **Noms qualifiés par une classe** : ils sont recherchés dans l'espace de noms de la classe (puis, s'il y a héritage entre classes [cf. p. 99], dans les espaces de noms des classes parentes en remontant celles-ci).
- **Noms qualifiés par une instance** : ils sont d'abord recherchés dans l'espace de noms de l'instance puis, s'ils n'y sont pas trouvés, ils sont recherchés dans l'espace de noms de la classe de l'instance (et, s'il y a héritage entre classes, dans les espaces de noms des classes parentes en remontant celles-ci).

Notons l'instruction `dir(x)`, qui, pour l'objet `x` donné en paramètre, fournit tous les noms qualifiés accessibles par cet objet, méthodes et variables membres compris, en suivant les règles d'accès en lecture (s'il y a de l'héritage, les espaces de noms des classes parentes sont aussi considérés).

Note : Les noms qualifiés ne sont jamais recherchés au niveau des modules par la règle LGI, ils suivent les règles d'accès définies par l'élément qui a servi à les qualifier.

Pour les accès aux attributs en écriture (affectation ou réaffectation)

- **Noms non qualifiés** : ils sont créés dans la portée locale courante.
- **Noms qualifiés par une classe** : ils sont créés dans l'espace de noms de la classe.
- **Noms qualifiés par une instance** : ils sont créés dans l'espace de noms de l'instance.

Masquage de noms

Un nom défini au niveau d'une instance peut masquer le même nom défini au niveau de la classe (sauf à passer explicitement par l'espace de noms de la classe).

L'aspect dynamique des espaces de noms en Python, où il est possible à tout moment de créer un attribut pour une classe ou une instance (*via* une méthode ou en accès direct à l'instance), peut amener à créer des bogues où par exemple une affectation faite par erreur au niveau d'une instance masque le même nom défini au niveau de la classe.

1. Les espaces de noms sont gérer par Python avec des dictionnaires.

Copyright © 2017 Dunod.

Exemple de masquage d'attribut :

```
class C:
    """Documentation de la classe C."""
    x = 23       # attribut de classe

a = C()      # a est un objet de la classe C (ou une instance)
b = C()      # b aussi
print(a.x)   # => 23 : affiche la valeur de l'attribut de l'instance a
a.x = 12     # modifie son attribut d'instance (attention...)
print(a.x)   # => 12 : pour l'objet a, x a une nouvelle valeur
print(C.x)   # => 23 : mais dans la classe la valeur n'a pas changé
print(b.x)   # => 23 : et b continue à utiliser celle de la classe
```

Important : pour modifier une variable de classe, il faut passer directement par cette classe. Dans notre exemple, si on veut modifier la variable de classe `x` de `C`, il faut directement écrire :

```
C.x = -1
```

8.4 Méthodes

Syntaxe

Une méthode s'écrit comme une fonction normale Python, on peut y utiliser tout ce que nous avons déjà vu (définition des paramètres, portée des variables, valeurs de retour, etc.), mais l'ensemble de la définition de la méthode, à partir du `def`, est indenté dans le corps de la classe.

Un élément important d'une méthode est son premier paramètre, `self` [1], obligatoire : il représente l'objet sur lequel la méthode sera appliquée.

Autrement dit, `self` est la référence d'instance [2].

```
>>> class C:
...     x = 23               # création d'un attribut de classe
...     def affiche(self):   # méthode affiche()
...         self.z = 42      # création d'un attribut d'instance
...         print(C.x)       # lecture des attributs de classe par la classe...
...         print(self.x)    # ... ou par l'instance
...         print(self.z)    # lecture des attributs d'instance par l'instance
...
>>> obj = C()
>>> obj.affiche()
23
23
4
```

Dans les méthodes, les règles d'accès aux noms qualifiés s'appliquent normalement, la référence d'instance `self`, reçue en premier paramètre, qualifiant l'objet manipulé [3].

L'appel aux méthodes se fait simplement en utilisant un nom qualifié à partir de l'objet manipulé. Python va rechercher la méthode dans l'espace de noms de l'objet suivant les règles que nous avons

1. Ce nom `self` n'est qu'une convention... mais elle est respectée par tous !
2. En Python cette référence est déclarée explicitement en premier paramètre de la méthode, dans d'autres langages elle existe implicitement sous un nom comme *this* ou *me*.
3. Il est possible de définir des méthodes dites *statiques*, qui ne reçoivent pas d'instance, en utilisant un « décorateur » `staticmethod` – voir les décorateurs [cf. p. 129].

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

déjà vues pour les attributs et, une fois celle-ci localisée, l'appeler en fournissant l'objet en premier paramètre.

Note : Il est d'ailleurs possible de passer directement par la classe de l'objet pour appeler une méthode, en fournissant soi-même directement l'objet en paramètre (`C.affiche(ojb)`).

8.5 Méthodes spéciales

Python offre un mécanisme qui permet d'enrichir les classes de caractéristiques supplémentaires : les *méthodes spéciales réservées*.

On pourra ainsi :

— **initialiser** l'objet instancié ;
— **modifier** son affichage ;
— **surcharger** (redéfinir) ses opérateurs.

Syntaxe

✎ Les méthodes spéciales portent des noms prédéfinis, précédés et suivis de **deux caractères de soulignement**.

8.5.1 L'initialiseur

Lors de l'instanciation d'un objet, la structure de base de l'objet est créée « nue » en mémoire, et la méthode `__init__` est automatiquement appelée pour initialiser l'objet. C'est typiquement dans cette méthode spéciale que sont créés les attributs d'instance avec leur valeur initiale.

```
>>> class C:
...     def __init__(self, n):
...         self.x = n     # initialisation de l'attribut d'instance x
...
>>> une_instance = C(42)  # paramètre obligatoire, affecté à n
>>> une_instance.x
42
```

C'est une *procédure* automatiquement invoquée lors de l'instanciation : elle ne retourne aucune valeur.

8.5.2 Surcharge des opérateurs

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de ses opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9          # addition entière
s = 'ab' + 'cd'    # concaténation
```

Python possède des méthodes de surcharge pour :

— tous les **types** (`__call__`, `__str__`, ...) ;
— les **nombres** (`__add__`, `__div__`, ...) ;
— les **séquences** (`__len__`, `__iter__`, ...).

Copyright © 2017 Dunod.

Soient deux instances, *obj1* et *obj2*, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes [1] :

Nom	Méthode spéciale	Utilisation
opposé	`__neg__`	`-obj1`
addition	`__add__`	`obj1 + obj2`
soustraction	`__sub__`	`obj1 - obj2`
multiplication	`__mul__`	`obj1 * obj2`
division	`__div__`	`obj1 / obj2`
division entière	`__floordiv__`	`obj1 // obj2`

8.5.3 Exemple de surcharge

Dans l'exemple suivant, nous surchargeons l'opérateur d'addition pour le type `Vecteur2D`. Nous surchargeons également la méthode spéciale `__str__` utilisée pour l'affichage par `print()`. Rappelons qu'il existe deux façons de formater un résultat : par `repr()` et par `str()`. La première est « pour la machine », la seconde « pour l'utilisateur ». `repr()` peut être redéfinie par `__repr__`.

```
class Vecteur2D:
    def __init__(self, x0, y0):
        self.x = x0
        self.y = y0

    def __add__(self, second):  # addition vectorielle
        return Vecteur2D(self.x + second.x, self.y + second.y)

    def __str__(self):          # affichage d'un Vecteur2D
        return "Vecteur({:g}, {:g})".format(self.x, self.y)

v1 = Vecteur2D(1.2, 2.3)
v2 = Vecteur2D(3.4, 4.5)
print(v1 + v2)   # Vecteur(4.6, 6.8)
```

Remarque : il est important de bien définir le sens des opérateurs lorsqu'une telle surcharge est mise en place. Il est parfois plus compréhensible d'avoir un appel de méthode explicite qu'un comportement lié à l'utilisation d'un opérateur à la sémantique peu claire. De la même façon, il est important de considérer l'opérateur vis-à-vis de l'instance dont la méthode est appelée : celle-ci doit-elle ou non voir ses attributs modifiés ?

Dans notre exemple d'addition, un nouveau vecteur est défini et retourné. Le vecteur `v1` en partie gauche de l'addition n'a pas de raison d'être modifié et ne l'est pas.

8.6 Héritage et polymorphisme

Un avantage décisif de la POO est qu'une classe peut toujours être spécialisée en une classe fille, qui *hérite* alors de tous les attributs (données et méthodes) de sa classe parente (ou « classe mère » ou « super classe »). Comme tous les attributs peuvent être redéfinis, deux méthodes de la classe fille et de la classe mère peuvent posséder le même nom mais effectuer des traitements différents (on parle

1. Pour plus de détails, consulter la documentation de référence du langage Python (*The Python language reference*) section 3, *Data model*, sous-section 3.3, *Special method names*.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

de « surcharge »). Du fait des règles sur les résolutions de noms qualifiés vues précédemment et du passage par un nom d'instance pour accéder à la méthode, il y a une adaptation dynamique de la méthode appelée à l'objet utilisé, et ce dès l'instanciation.

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le *polymorphisme* permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il appelle une méthode sur un objet, le type précis de l'objet sur lequel celle-ci va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode *via* sa classe ou une des classes héritées.

8.6.1 Formalisme de l'héritage et du polymorphisme

Définition

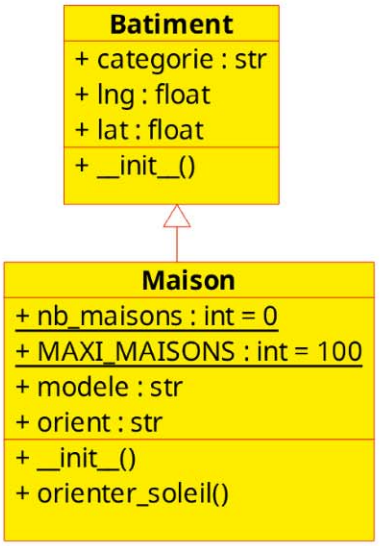
L'**héritage** est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.

Le **polymorphisme par dérivation** est la faculté pour deux méthodes (ou plus) portant le même nom, mais appartenant à des classes héritées distinctes, d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

Syntaxe

Lors de la définition d'une nouvelle classe, on indique la classe héritée entre parenthèses juste après le nom de la nouvelle classe (si celle-ci hérite de plusieurs classes parentes, on les sépare par des virgules).

En modélisation UML, on représente l'héritage par une flèche à la pointe vide, de la classe fille vers la classe mère. Si l'on reprend notre exemple du début du chapitre, en l'étendant pour pouvoir gérer à terme plusieurs catégories de bâtiments (habitat, hôpital, école, commerce...) dans notre projet urbanistique, cela donne :



Dans le code Python, cela se représente donc par :

```
class Batiment:
    ...
```

Copyright © 2017 Dunod.

```
class Maison(Batiment):
    ...
```

Mais c'est encore incomplet, il faut en effet s'assurer que toutes les méthodes d'initialisation sont bien appelées lorsqu'une nouvelle `Maison` est créée, afin de construire chaque classe dont dépend l'objet avec son code d'initialisation propre. Pour cela, Python dispose de la fonction spéciale `super()`, qui permet d'appeler une méthode située dans une classe héritée sans avoir à préciser le nom de celle-ci.

```
class Batiment:
    """Définition d'un bâtiment en général."""
    def __init__(self, categorie, longitude, latitude):
        self.categorie = categorie
        self.lng = longitude
        self.lat = latitude

class Maison(Batiment):
    """Définition d'une maison."""
    nb_maisons = 0
    MAXI_MAISONS = 100

    def __init__(self, modele, longitude, latitude, orientation):
        """Initialiseur de Maison."""
        if Maison.nb_maisons >= Maison.MAXI_MAISONS:
            raise RuntimeError("Trop de maisons consruites")
        super().__init__("habitat", longitude, latitude)
        self.modele = modele
        self.orient = orientation
        Maison.nb_maisons += 1

    def orienter_soleil(self):
        """Changement du sens d'orientation de la maison vers le sud."""
        self.orient = "sud"
```

Dans le `__init__()` de `Maison`, la ligne `super().__init__()` permet d'appeler l'initialiseur de la classe parente `Batiment.__init__()` [1], Python se chargeant d'identifier la classe parente – notre exemple est simple, mais Python autorise l'héritage multiple et permet l'héritage dit « en diamant », et dans ces cas compliqués il vaut mieux lui laisser faire l'appel des méthodes d'initialisation dans le bon ordre en utilisant la fonction `super()`.

Remarque : en Python, toutes les classes héritent par défaut de la classe `object`, et donc des méthodes spéciales qui y sont définies.

8.6.2 Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe `QuadrupedeDebout` hérite de la classe mère `Quadrupede`, et la méthode `piedsAuContactDuSol()` est polymorphe :

```
class Quadrupede:
    def piedsAuContactDuSol(self):
        return 4
```

1. Dans d'autres langages, l'appel des initialiseurs (on les appelle généralement « constructeurs ») des classes parentes se fait de façon implicite avant d'appeler celui de la classe en cours ; en Python cet appel doit être fait explicitement.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
class QuadrupedeDebout(Quadrupede):
    def piedsAuContactDuSol(self):
        return 2

chat = Quadrupede()
chat.piedsAuContactDuSol()   # affiche : 4
homme = QuadrupedeDebout()
homme.piedsAuContactDuSol()  # affiche : 2
```

8.7 Notion de « conception orientée objet »

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles en utilisant l'*association* ou la *dérivation*.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

8.7.1 Relation, association

Définition

✍ Une **association** représente un lien unissant les instances de classes. On parlera d'une association entre deux classes si les deux classes correspondent à des entités pouvant être en relation mais pouvant aussi exister séparément, par exemple un étudiant dans un cours.

On parlera d'une **relation d'agrégation** si l'association est liée au fait qu'un objet d'une classe (l'agrégat) a dans sa constitution un ou plusieurs objets d'une autre classe (les « composites »), par exemple une roue de voiture qui comporte un pneu (lors de l'analyse on trouve souvent ces relations dans des expressions « a-un » ou « utilise-un »). Lorsque, dans l'agrégation, l'objet composite n'existe que par l'existence de l'agrégat auquel il appartient, on parlera plus précisément de composition.

Note : Il faut bien faire attention, lors de l'analyse et de la modélisation, à se restreindre au problème à résoudre et à ne pas chercher à représenter le monde dans toute sa complexité. Dans l'exemple des roues de voiture, si on se place dans l'optique d'un monteur de pneus, alors pneus et jantes doivent pouvoir exister indépendamment au même niveau et on aura plutôt une relation d'instance.

En UML, on schématise une association de classes par un trait reliant ces deux classes. On place autour de ce trait diverses indications textuelles : dénomination de la relation, attributs par lesquels elle sera accessible, multiplicités… Pour une agrégation, on place un losange vide du côté de la « classe utilisatrice », l'agrégat (et si la relation est une composition, le losange sera alors plein).

Si, dans notre exemple urbanistique, on désire séparer dans une classe spécifique les composantes des coordonnées des bâtiments pour y regrouper une série de fonctionnalités dont on a besoin par ailleurs (calculs de distance, recherche de l'altitude dans une base de données…), on schématisera de la façon suivante, où l'attribut `coord` du `Batiment` est devenu une relation de composition vers la classe `Coord` (☞ Fig. 8.1).

L'implémentation Python utilisée est généralement l'intégration d'autres objets dans le constructeur de la classe conteneur, dans notre exemple :

```
class Coord:
    """Définition de coordonnées géodésiques (sans la hauteur)."""
    def __init__(self, lng, lat):
```

Copyright © 2017 Dunod.

```
        self.lng = lng
        self.lat = lat

class Batiment:
    """Définition d'un bâtiment en général."""
    def __init__(self, categorie, longitude, latitude):
        self.categorie = categorie
        self.coord = Coord(longitude, latitude)
```

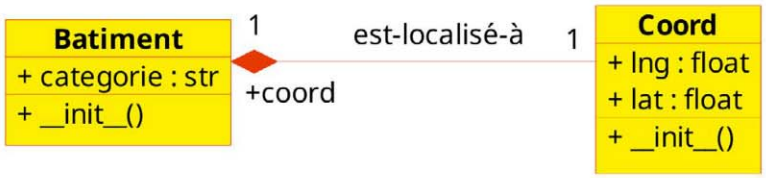


FIGURE 8.1 – Une association peut être étiquetée et avoir des multiplicités

Définition

✍ Une **agrégation** est une association non symétrique entre deux classes (l'*agrégat* et le *composant*).

Définition

✍ Une **composition** est un type particulier d'agrégation dans lequel la vie des composants est liée à celle de l'agrégat.

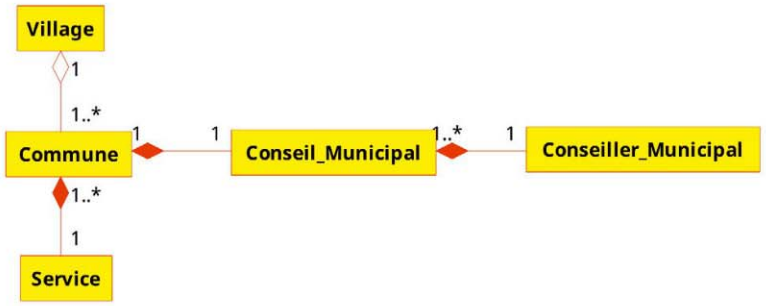


FIGURE 8.2 – On peut mêler les deux types d'associations

La disparition de l'agrégat `Commune` entraîne la disparition des composants `Services` et `Conseil_Municipal` ainsi que `Conseiller_Municipal`, alors que `Village` n'en dépend pas et peut continuer à exister.

8.7.2 Dérivation

Définition

✍ La **dérivation** décrit la création de sous-classes par spécialisation. Elle repose sur la relation « est-un ».

Dans notre exemple précédent, nous avons créé une classe `Maison` dérivant de la classe `Batiment` afin de la spécialiser, puis une classe `Coord` dédiée à la gestion de coordonnées géodésiques. Nous pourrions étendre notre modèle à d'autres types de bâtiments, avec leurs spécificités, pouvant même avoir des relations entre eux.

Pour réaliser la dérivation en Python, on utilise simplement le mécanisme déjà vu de l'héritage.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

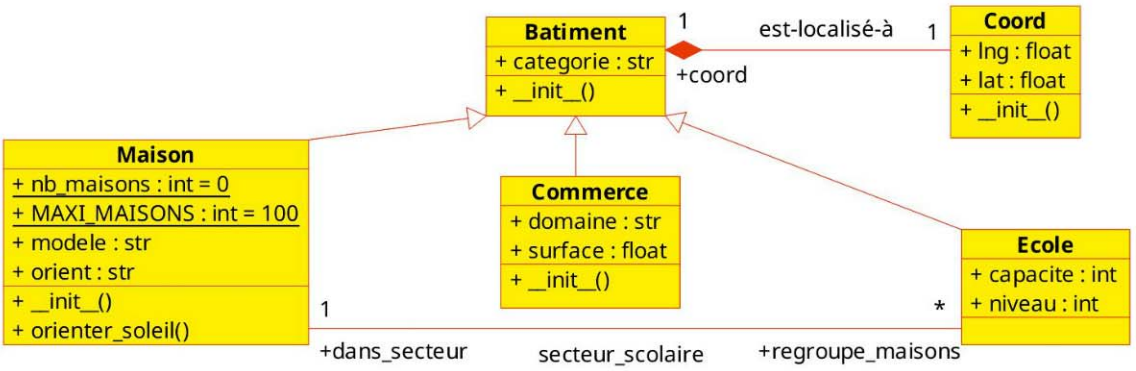


FIGURE 8.3 – Dérivations à partir de la classe `Maison`

Dans l'exemple suivant, un `Carre` « est-un » `Rectangle` particulier pour lequel on appelle l'initialiseur de la classe mère avec les paramètres `longueur=cote` et `largeur=cote`. Il faut faire attention au moment où l'initialiseur de la classe parente est appelé, de façon à ce que les modifications effectuées par la sous-classe (ici la surcharge de l'attribut `nom`) ne soient pas écrasées :

```
class Rectangle:
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l = longueur, largeur
        self.nom = "rectangle"
    def __str__(self):
        return "nom : {}".format(self.nom)

class Carre(Rectangle):     # héritage simple
    """Sous-classe spécialisée de la super-classe Rectangle."""
    def __init__(self, cote=20):
        # appel au constructeur de la super-classe de Carre :
        super().__init__(cote, cote)
        self.nom = "carré" # surcharge d'attribut

r = Rectangle()
c = Carre()
print(r)  # affiche : nom : rectangle
print(c)  # affiche : nom : carré
```

8.8 Exercice

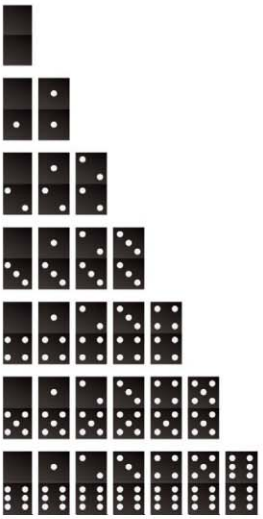
1. ✠ Un domino est une pièce constituée de deux extrémités comportant chacune un dessin de zéro (vide) à six points. Un jeu de dominos comprend 28 pièces composées des combinaisons des valeurs ci-dessus. L'objectif est d'apposer sur la table les pièces en appariant les extrémités de même valeur. Voir le détail des règles sur `https://fr.wikipedia.org/wiki/Dominos_(jeu)`.

   Créer une classe `Domino` dont chaque instance (domino) a deux attributs correspondant aux valeurs de ses deux extrémités (que l'on fournira à la construction). Pour cette classe, créer une méthode `appariement` qui permet d'évaluer si un domino peut être apparié par une de ses extrémités avec un autre domino. Si l'appariement est possible, la méthode renvoie la valeur de l'extrémité par laquelle il peut se faire. S'il n'est pas possible, la méthode renvoie `None`.

Copyright © 2017 Dunod.

Définir une liste de pioche contenant l'ensemble des dominos. Mélanger celle-ci à l'aide de la méthode `shuffle()` du module `random`. Prendre les sept premiers dominos de la pioche pour le joueur 1, et les sept suivants pour le joueur 2 (supprimer ces dominos de la pioche).

Pour le premier domino du joueur 1, afficher tous les dominos du joueur 2 qui peuvent être appariés.

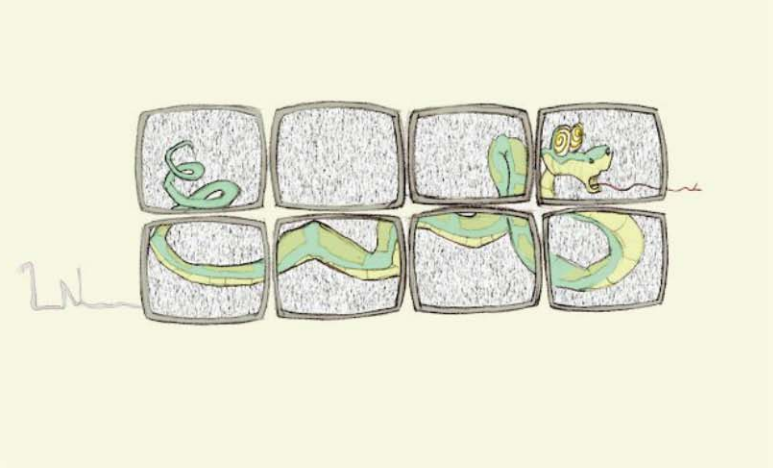


Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Chapitre 9

La programmation graphique orientée objet



Hégémoniques dans les interfaces avec les utilisateurs et donc dans les applications, les *interfaces graphiques* sont programmables en Python.

Parmi les différentes bibliothèques graphiques utilisables dans Python (`GTK+`, `wxWidgets`, `Qt`...), la bibliothèque `tkinter`, issue du langage `tcl/Tk`, est installée de base dans toutes les distributions Python. `tkinter` facilite la construction d'interfaces graphiques simples.

Après avoir importé la bibliothèque, la démarche consiste à créer, configurer et positionner les éléments graphiques (widgets) utilisés, à définir les fonctions/méthodes associées aux widgets, puis à entrer dans une boucle chargée de récupérer et traiter les différents événements pouvant se produire au niveau de l'interface graphique : interactions de l'utilisateur, besoins de mises à jour graphiques, etc.

9.1 Programmes pilotés par des événements

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une *boucle d'événements* (☞ Fig. 9.1).

9.2 La bibliothèque `tkinter`

9.2.1 Présentation

C'est une bibliothèque issue de l'extension graphique, `Tk`, du langage `Tcl` [1]. Cette extension a largement essaimé hors de `Tcl/Tk` et on peut l'utiliser en Perl, Python, Ruby, etc. Dans le cas de Python 3, l'extension a été nommée `tkinter`.

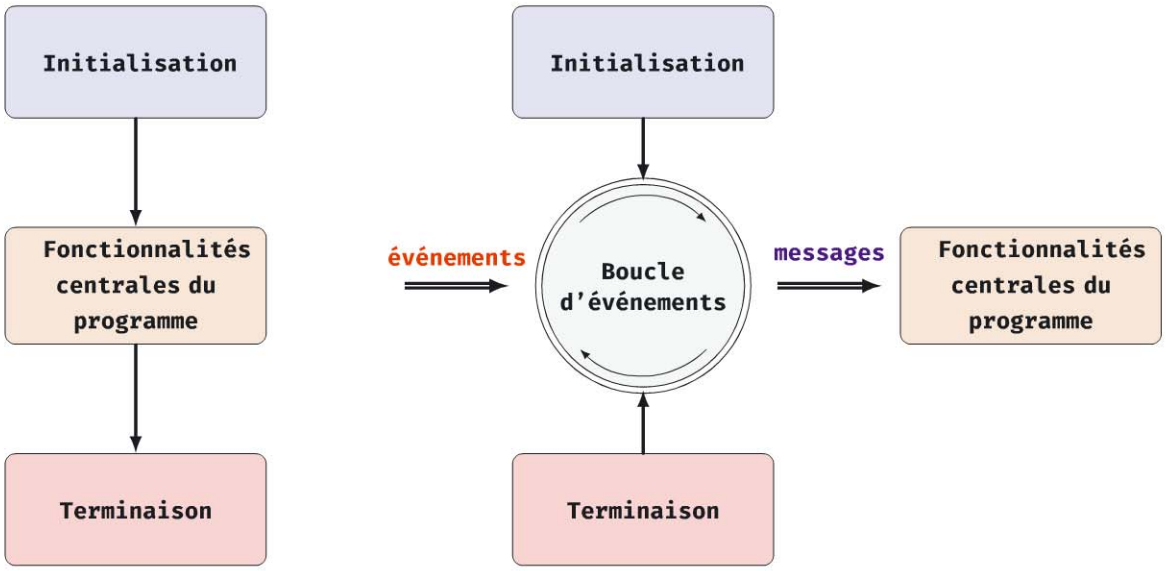
1. Langage développé en 1988 par John K. Ousterhout de l'Université de Berkeley.

Copyright © 2017 Dunod.

Parallèlement à `Tk`, d'autres extensions ont été développées dont certaines sont utilisées en Python. Par exemple, le module standard `Tix` met une quarantaine de composants graphiques à la disposition du développeur.

De son côté, le langage `Tcl/Tk` a largement évolué. La version 8.5 actuelle offre une bibliothèque appelée `Ttk` qui permet d'« habiller » les composants avec différents thèmes ou styles. Ce module est également disponible à partir de Python 3.1.1.

Une documentation très complète (en anglais) de tkinter 8.5 est disponible sur le site de l'université New Mexico Tech [1].



(a) Séquentielle (b) Pilotée par une boucle d'événements

FIGURE 9.1 – Deux styles de programmations

Un exemple `tkinter` simple (☞ Fig. 9.2)

```
import tkinter

# création d'un widget affichant un simple message textuel
widget = tkinter.Label(None, text='Bonjour monde graphique !')
widget.pack()       # positionnement du label
widget.mainloop()   # lancement de la boucle d'événements
```

FIGURE 9.2 – Un exemple simple : l'affichage d'un `Label`

1. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

Copyright © 2017 Dunod.

9.2.2 Les widgets de `tkinter`

Remarque

✓ On appelle *widgets* (mot valise, contraction de *window* et *gadget*) les composants graphiques de base d'une bibliothèque.

Liste des principaux widgets de tkinter :

- **Tk** : fenêtre de plus haut niveau ;
- **Frame** : contenant pour organiser d'autres widgets ;
- **LabelFrame** : contenant pour organiser d'autres widgets, avec un cadre et un titre ;
- **Spinbox** : un widget de sélection multiple parmi une liste de valeurs ;
- **Label** : zone de texte fixe (étiquette, message...) ;
- **Message** : zone d'affichage multiligne ;
- **Entry** : zone de saisie ;
- **Text** : édition de texte simple ou multiligne ;
- **ScrolledText** : widget `Text` avec ascenseur ;
- **Button** : bouton d'action avec texte ou image ;
- **Checkbutton** : bouton à deux états (case à cocher) ;
- **Radiobutton** : bouton à deux états, un seul actif par groupe de boutons radio ;
- **Scale** : glissière à plusieurs positions ;
- **PhotoImage** : sert à placer des images (GIF et PPM/PGM) sur des widgets ;
- **Menu** : menu déroulant associé à un `Menubutton` ;
- **Menubutton** : bouton ouvrant un menu d'options ;
- **OptionMenu** : liste déroulante ;
- **Scrollbar** : ascenseur ;
- **Listbox** : liste à sélection pour des textes ;
- **Canvas** : zone de dessins graphiques ou de photos ;
- **PanedWindow** : interface à onglets.

9.2.3 Le positionnement des widgets

Là où certaines bibliothèques d'interface graphiques procèdent par positionnement absolu des éléments, `tkinter` utilise un mécanisme permettant de décrire des positionnements relatifs suivant différentes politiques de dimensionnement et de placement. Ceci permet d'adapter les widgets à leur contenu (par exemple lorsque l'on traduit une interface graphique dans une autre langue), au périphérique d'affichage et à sa résolution, ainsi qu'au redimensionnement des fenêtres.

`tkinter` possède trois gestionnaires de positionnement :

- le **packer** dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux, en suivant une politique paramétrable ;
- le **gridder** dimensionne et positionne chaque widget dans une ou plusieurs cellules d'un tableau défini dans un widget conteneur ;
- le **placer** dimensionne et place chaque widget dans un widget conteneur selon l'espace explicitement demandé. C'est un placement absolu (usage peu fréquent avec tkinter).

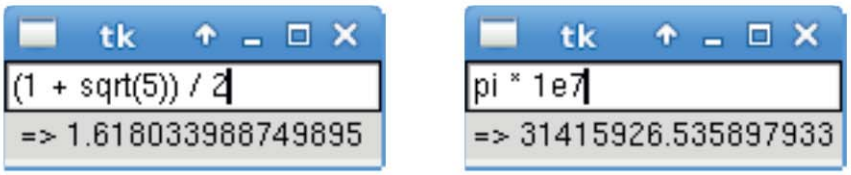
Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

9.3 Deux exemples

9.3.1 Une calculette

Cette application [1] implémente une calculette minimaliste, mais complète.

```
from tkinter import *
from math import *

def evaluer(event):
    chaine.configure(text = '=> ' + str(eval(entree.get())))

# Programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fenetre = Tk()
entree = Entry(fenetre)
entree.bind('<Return>', evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```



(a) φ, le nombre d'or

(b) Approximation du nombre de secondes en un an

FIGURE 9.3 – Exemple d'utilisation de la calculette

Le programme principal se compose de l'instanciation d'une fenêtre `Tk()` contenant un widget nommé `entree` de type `Entry()`, pour effectuer la saisie, et un widget nommé `chaine` de type `Label()`, pour afficher le résultat. Le positionnement de ces deux widgets est assuré à l'aide de la méthode `pack()`.

L'appui sur la touche `Entrée` dans le champ de saisie est associé à un appel à la fonction `evaluer()` grâce à l'utilisation de la méthode `bind()` du widget `entree` avec l'événement noté `<Return>`.

Enfin on démarre l'interaction en activant la boucle d'événements avec un appel à la méthode `mainloop()`.

Lors de l'appui sur `Entrée`, la fonction `evaluer()` est automatiquement appelée par `tkinter` ; elle récupère le texte du champ `entree`, utilise la fonction standard Python `eval()` pour évaluer ce texte comme s'il s'agissait d'une expression Python (dans le contexte des noms importés), et place le résultat de cette évaluation sous forme de texte dans le widget `chaine`.

9.3.2 tkPhone

On se propose de créer un script de gestion d'un carnet téléphonique. L'aspect de l'application est illustré ☞ Fig. 9.4.

1. Exemple adapté de [6], p. 217.

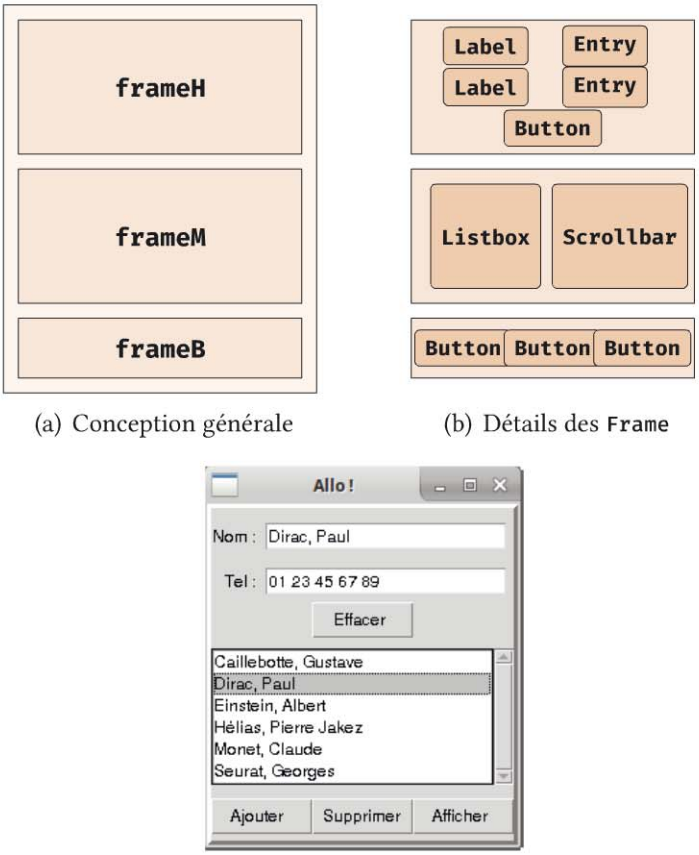
Copyright © 2017 Dunod.

Notion de *callback*

Nous avons vu que la programmation d'interface graphique passe par une boucle principale chargée de traiter les différents événements qui se produisent.

Cette boucle est généralement gérée directement par la bibliothèque d'interface graphique utilisée, il faut donc pouvoir spécifier à cette bibliothèque quelles fonctions doivent être appelées dans quels cas. Ces fonctions sont nommées des *callbacks* (ou rappels) car elles sont appelées directement par la bibliothèque d'interface graphique lorsque des événements spécifiques se produisent. Dans l'exemple précédent, l'association événement/callback a été réalisée par l'instruction :

```
entree.bind('<Return>', evaluer).
```



(a) Conception générale (b) Détails des `Frame`

(c) L'application

FIGURE 9.4 – tkPhone

Conception graphique

La conception graphique va nous aider à choisir les bons widgets.

En premier lieu, il est prudent de commencer par une conception manuelle ! En effet rien ne vaut un papier, un crayon et une gomme (ou encore un tableau) pour se faire une idée de l'aspect que l'on veut obtenir.

Dans notre cas, on peut concevoir trois zones :

1. Une zone supérieure, dédiée à l'affichage.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

2. Une zone médiane contenant une liste alphabétique ordonnée.
3. Une zone inférieure, formée de boutons de gestion de la liste placée au-dessus.

Chacune de ces zones est codée par une instance de `Frame()`. Elles sont positionnées l'une sous l'autre grâce au `packer`, et toutes trois sont incluses dans une instance de `Tk()` (cf. conception ☞ Fig. 9.4).

Le code de l'interface graphique

Méthodologie : on se propose de séparer le codage de l'interface graphique de celui des *callbacks*. Pour cela on utilise l'héritage entre une classe parente chargée de gérer l'aspect graphique et une classe enfant chargée de gérer l'aspect fonctionnel de l'application, contenu dans les callbacks. Comme nous l'avons vu précédemment (cf. § 8.7 p. 102), c'est un cas de polymorphisme de dérivation.

Cette méthode est très couramment utilisée dans les logiciels de construction d'interface graphique, qui se chargent de générer complètement le module de la classe parente (et de le régénérer en totalité ou en partie en cas de modification de l'interface) et qui laissent l'utilisateur placer son code dans le module de la classe fille.

Voici donc dans un premier temps le code de l'interface graphique. L'initialiseur crée la fenêtre de base `root` et appelle la méthode `construireWidgets()`. Celle-ci suit la conception graphique exposée ci-dessus (☞ Fig. 9.4) et remplit chacun des trois `frames`. Les options *ad hoc* des gestionnaires de positionnement ont été utilisées pour s'assurer du bon comportement des widgets en cas de redimensionnement de la fenêtre. Au niveau esthétique, nous avons utilisé le module `tkinter.ttk` qui redéfinit certains widgets de base pour qu'ils aient un aspect standard suivant la plateforme sur laquelle est exécuté le programme.

Un ensemble de méthodes est mis en place afin de permettre de manipuler l'interface (lecture / modification des valeurs et/ou des caractéristiques des widgets), sans avoir à connaître les détails de celle-ci (noms des variables, types des widgets…) [1]. Ceci permet ultérieurement de modifier l'interface au niveau de la classe parent sans avoir à modifier la « logique métier » qui est dans la classe enfant [2].

Le travail sur l'esthétique de l'interface graphique, le respect des normes et conventions auxquelles l'utilisateur s'attend suivant la plateforme utilisée, mais aussi la logique du comportement des widgets (gestion du « focus », désactivation des widgets qui ne sont pas utilisables, bulles d'aide, signalisation des saisies invalides dès que possible…) sont très importants dans une application car, au-delà du bon fonctionnement de la logique métier, ce sont les aspects auxquels l'utilisateur est immédiatement et directement confronté. Ceci demande du temps de développement, souvent des essais et corrections, un savoir-faire qui vient avec l'expérience (de développeur mais aussi d'utilisateur) et la lecture – le premier volume des guide de développement Apple « Inside Macintosh » commençait par un chapitre entier de conseils et de normes à respecter pour le développement d'interfaces graphiques.

Les callbacks sont quasi vides (levée d'une exception `raise NotImplementedError` afin d'éviter un appel de méthode que la sous-classe aurait oublié de redéfinir – on peut faire le choix de placer simplement une instruction `pass` pour permettre d'appeler ces callbacks lors des tests sans que cela n'ait de conséquence.

Comme pour tout bon module, un auto-test permet de vérifier le bon fonctionnement (ici le bon aspect) de l'interface :

1. Le « Modèle-Vue-Contrôleur » ou MVC est une autre façon standard de réaliser cette séparation. `tkinter` fournit des types variables (`StringVar`, `DoubleVar`, etc.) qui permettent aussi ce découpage.
2. C'est une conception idéale vers laquelle il faut tendre, mais qui n'est pas toujours aisée à mettre en œuvre lorsque l'interaction avec l'utilisateur est riche et entraîne une intrication entre la logique et la présentation.

Copyright © 2017 Dunod.

```
# -*- coding: utf-8 -*-

# Doc tkinter très complète:
# => http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

# import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

# définition de classe ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class Allo_IHM:
    """IHM de l'application 'répertoire téléphonique'."""
    def __init__(self):
        """Initialiseur/lanceur de la fenêtre de base."""
        self.root = tk.Tk()     # Fenêtre de l'application
        self.root.title("Allo !")
        self.root.config(relief=tk.RAISED, bd=3)
        self.construireWidgets()

    def boucleEnevementielle(self):
        self.champsNom.focus()
        self.root.mainloop()

    def construireWidgets(self):
        """Configure et positionne les widgets."""
        # frame "valeursChamps" (en haut avec bouton d'effacement)
        frameH = ttk.Frame(self.root)
        frameH.pack(fill=tk.X)
        frameH.columnconfigure(1, weight=1)

        ttk.Label(frameH, text="Nom :").grid(row=0, column=0, sticky=tk.E)
        self.champsNom = tk.Entry(frameH)
        self.champsNom.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=10)

        ttk.Label(frameH, text="Tel :").grid(row=1, column=0, sticky=tk.E)
        self.champsTel = tk.Entry(frameH)
        self.champsTel.grid(row=1, column=1, sticky=tk.EW, padx=5, pady=2)

        b = ttk.Button(frameH, text="Effacer ", command=self.effaceChamps)
        b.grid(row=2, column=0, columnspan=2, pady=3)

        # frame "liste" (au milieu)
        frameM = ttk.Frame(self.root)
        frameM.pack(fill=tk.BOTH, expand=True)

        self.scroll = ttk.Scrollbar(frameM)
        self.listeSelection = tk.Listbox(frameM, yscrollcommand=self.scroll.set,
                                         height=6)
        self.scroll.config(command=self.listeSelection.yview)
        self.scroll.pack(side=tk.RIGHT, fill=tk.Y, pady=5)
        self.listeSelection.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, pady=5)
        self.listeSelection.bind("<Double-Button-1>",
                    lambda event: self.cb_afficher())

        # frame "boutons" (en bas)
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
        frameB = ttk.Frame(self.root)
        frameB.pack(pady=3, side=tk.BOTTOM, fill=tk.NONE)

        b1 = ttk.Button(frameB, text="Ajouter  ", command=self.cb_ajouter)
        b2 = ttk.Button(frameB, text="Supprimer", command=self.cb_supprimer)
        b3 = ttk.Button(frameB, text="Afficher ", command=self.cb_afficher)
        b1.pack(side=tk.LEFT, pady=2)
        b2.pack(side=tk.LEFT, pady=2)
        b3.pack(side=tk.LEFT, pady=2)

    # Méthodes d'échange d'informations application <==> GUI
    def majListeSelection(self, lstnoms):
        """Remplissage complet de la liste à sélection avec les noms."""
        self.listeSelection.delete(0, tk.END)
        for nom in lstnoms:
            self.listeSelection.insert(tk.END, nom)

    def indexSelection(self):
        """Retourne le n° de la ligne actuellement sélectionnée."""
        return int(self.listeSelection.curselection()[0])

    def changeChamps(self, nom, tel):
        """Modification des affichages dans les champs de saisie."""
        self.champsNom.delete(0, tk.END)
        self.champsNom.insert(0, nom)
        self.champsTel.delete(0, tk.END)
        self.champsTel.insert(0, tel)
        self.champsNom.focus()

    def effaceChamps(self):
        """Effacement des champs de saisie."""
        self.changeChamps('', '')

    def valeursChamps(self):
        """Retourne la saisie nom/tél actuelle."""
        nom = self.champsNom.get()
        tel = self.champsTel.get()
        return nom, tel

    def alerte(self, titre, message):
        """Affiche un message à l'utilisateur."""
        messagebox.showinfo(titre, message)

    # Méthodes à redéfinir dans l'application (actions liées aux boutons)
    def cb_ajouter(self):
        """Ajout dans la liste du contenu des champs de saisie."""
        raise NotImplementedError("cb_ajouter à redéfinir")

    def cb_supprimer(self):
        """Suppression de la liste de l'entrée des champs de saisie."""
        raise NotImplementedError("cb_supprimer à redéfinir")

    def cb_afficher(self):
        """Affichage dans les champs de saisie de la sélection."""
        raise NotImplementedError("cb_afficher à redéfinir")
```

Copyright © 2017 Dunod.

```
# auto-test ==========================================================================
if __name__ == '__main__':
    # instancie l'IHM, callbacks inactifs
    app = Allo_IHM()
    app.boucleEnevementielle()
```

Le code de l'application

Nous allons utiliser le module de la partie interface graphique de la façon suivante :

- on importe la classe `Allo_IHM` depuis le module précédent ;
- on crée une classe `Allo` qui en dérive ;
- son initialiseur appelle l'initialiseur de la classe de base pour hériter de toutes ses caractéristiques et bénéficier de l'interface graphique. Il définit ensuite les variables membres nécessaires à la gestion du carnet d'adresses et charge le fichier de données qui lui a été fourni en paramètre. Enfin il appelle la méthode de l'interface graphique chargée d'afficher les données ;
- on place dans des méthodes séparées et identifiées ce qui est lié à la gestion du fichier de données ;
- il reste à surcharger les callbacks, ce qui se limite à des appels aux méthodes de l'interface graphique pour récupérer les saisies ou modifier les affichages, à des mises à jour de la liste stockée en mémoire, et à des appels aux méthodes sur le fichier.

Enfin, le script instancie l'application et démarre la boucle événementielle :

```
# -*- coding: utf-8 -*-

# tkPhone.py

# import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from collections import namedtuple
from os.path import isfile
from tkPhone_IHM import Allo_IHM

SEPARATEUR = '\t'
LigneRep = namedtuple("LigneRep", "nom tel")

# définition de classe ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class Allo(Allo_IHM):
    """Répertoire téléphonique."""

    def __init__(self, fic='phones.txt'):
        super().__init__()      # => constructeur de l'IHM classe parente.
        self.phoneList = []     # Liste des (nom,tél) à gérer.
        self.fic = ""
        self.chargerFichier(fic)

    def chargerFichier(self, nomfic):
        """Chargement de la liste à partir d'un fichier répertoire."""
        self.fic = nomfic       # Mémorise le nom du fichier
        self.phoneList = []     # Repart avec liste vide.
        if isfile(self.fic):
            with open(self.fic, encoding="utf8") as f:
                for line in f:
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

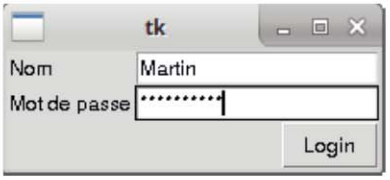
```
                    nom, tel, *reste = line[:-1].split(SEPARATEUR)[:2]
                    self.phoneList.append(LigneRep(nom, tel))
        else:
            with open(self.fic, "w", encoding="utf8"):
                pass
        self.phoneList.sort()
        self.majListeSelection([x.nom for x in self.phoneList])

    def enregistrerFichier(self):
        """Enregistre l'ensemble de la liste dans le fichier."""
        with open(self.fic, "w", encoding="utf8") as f:
            for i in self.phoneList:
                f.write("%s%s%s\n" % (i.nom, SEPARATEUR, i.tel))

    def ajouterFichier(self, nom, tel):
        """Ajoute un enregistrement à la fin du fichier."""
        with open(self.fic, "a", encoding="utf8") as f:
            f.write("%s%s%s\n" % (nom, SEPARATEUR, tel))

    def cb_ajouter(self):
        # maj de la liste
        nom, tel = self.valeursChamps()
        nom = nom.replace(SEPARATEUR, ' ')  # Sécurité
        tel = tel.replace(SEPARATEUR, ' ')  # Sécurité
        if (nom == "") or (tel == ""):
            self.alerte("Erreur", "Il faut saisir nom et n° de téléphone.")
            return
        self.phoneList.append(LigneRep(nom, tel))
        self.phoneList.sort()
        self.majListeSelection([x.nom for x in self.phoneList])
        self.ajouterFichier(nom, tel)
        self.effaceChamps()

    def cb_supprimer(self):
        # maj de la liste
        nom, tel = self.phoneList[self.indexSelection()]
        self.phoneList.remove(LigneRep(nom, tel))
        self.majListeSelection([x.nom for x in self.phoneList])
        self.enregistrerFichier()
        self.effaceChamps()

    def cb_afficher(self):
        nom, tel = self.phoneList[self.indexSelection()]
        self.changeChamps(nom, tel)

# programme principal ================================================
app = Allo()   # instancie l'application
app.boucleEnevementielle()
```

Copyright © 2017 Dunod.

9.4 Exercices

1. ✠ Écrivez un module Python utilisant `tkinter` et permettant de construire une interface de dialogue contenant un label « Valeur : » suivi d'un champ de saisie, en dessous duquel on trouve un bouton case à cocher associée au texte « Toujours utiliser cette valeur », et encore en dessous deux boutons « Ok » et « Annuler ».

2. ✠ Écrivez une interface `tkinter` de saisie du nom et du mot de passe d'un utilisateur. Ajoutez un bouton « Login » qui quitte l'application.



Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Chapitre 10

Quelques techniques avancées de programmation

Ce chapitre présente de nombreux exemples de techniques avancées dans les trois paradigmes que supporte Python : les programmations procédurale, objet et fonctionnelle.

10.1 Techniques procédurales

10.1.1 Le pouvoir de l'introspection

C'est l'un des atouts de Python. On entend par *introspection* la possibilité d'obtenir, à l'exécution, des informations sur les objets manipulés par le langage.

L'aide en ligne

Les shells Python des outils de `Pyzo` offrent la commande magique `?` qui permet, grâce à l'introspection, d'avoir directement accès à l'autodocumentation sur une commande (par exemple `?print`). L'outil `Pyzo` « Interactive help » fournit une zone dédiée à cette aide, avec une mise en forme plus avancée, et prenant directement en compte la dernière saisie de l'utilisateur, qu'elle soit dans l'éditeur de texte ou dans un shell Python.

Il existe aussi une commande magique `??` dans les shells Python de `Pyzo`, qui donne un accès direct à la fonction `pydoc.help()` permettant de naviguer parmi l'ensemble des chaînes de documentation incluses dans les modules Python (cette fonction est généralement disponible aussi directement avec son nom `help()`).

Enfin l'éditeur de `Pyzo` fournit une aide très efficace sous forme d'une bulle d'aide s'affichant automatiquement à chaque ouverture d'une fonction.

Copyright © 2017 Dunod.

La fonction utilitaire `printInfo()`, dont le code est présenté ci-dessous, est un exemple d'utilisation des capacités d'introspection de Python : elle filtre, parmi les attributs de son argument, ceux qui sont des méthodes (exécutables), dont le nom ne commence pas par « _ », et affiche leur *docstring* sous une forme plus lisible que `help()` :

```
def printInfo(object):
    """Filtre les méthodes disponibles de <object>."""
    methods = [method for method in dir(object)
        if callable(getattr(object, method)) and not method.startswith('_')]

    for method in methods:
        print(getattr(object, method).__doc__)
```

Par exemple, l'appel `printInfo([])` affiche la documentation :

```
L.append(object) -- append object to end
L.count(value) -> integer -- return number of occurrences of value
L.extend(iterable) -- extend list by appending elements from the iterable
L.index(value, [start, [stop]]) -> integer -- return first index of value.
Raises ValueError if the value is not present.
L.insert(index, object) -- insert object before index
L.pop([index]) -> item -- remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.
L.remove(value) -- remove first occurrence of value.
Raises ValueError if the value is not present.
L.reverse() -- reverse *IN PLACE*
L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
```

Les fonctions `type()`, `dir()` et `id()`

Ces fonctions fournissent respectivement le type, les noms définis dans l'espace de noms et l'identification (unique) d'un objet (en `CPython`, cette identification est la localisation en mémoire) :

```
>>> li = [1, 2, 3]
>>> type(li)
<class 'list'>
>>> dir(li)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__','__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__','__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__','append', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> id(li)
3074801164
```

Les fonctions `locals()` et `globals()`

Comme nous l'avons déjà vu [cf. p. 66], ces fonctions retournent les dictionnaires des noms locaux et globaux au moment de leur appel, et permettent ainsi de découvrir à l'exécution l'ensemble des noms des variables, fonctions, classes… présents.

Copyright © 2017 Dunod.

Le module sys

Ce module fournit nombre d'informations générales concernant le système utilisé, entre autres le chemin du programme exécutable de l'interpréteur Python, la plateforme informatique où il s'exécute (le module `platform` fournit plus de détails sur celle-ci), la version de Python utilisée, les arguments fournis au processus lors de l'appel (`argv`, « arguments ligne de commande »), la liste des chemins dans lesquels les modules Python sont recherchés, le dictionnaire des modules chargés... :

```
>>> import sys
>>> sys.executable
'/usr/bin/python3'
>>> sys.platform
'linux2'
>>> sys.version
'3.6.0 |Continuum Analytics, Inc.| (default, Dec 23 2016, 12:22:00) \n[GCC 4.4.7 20120313 (Red Hat
    4.4.7-1)]'
>>> sys.argv
['']
>>> sys.path
['', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linux2', '/usr/lib/python3.2/lib-dynload', '/
    usr/local/lib/python3.2/dist-packages', '/usr/lib/python3/dist-packages']
>>> sys.modules
{'reprlib': <module 'reprlib' from '/usr/lib/python3.2/reprlib.py'>, 'heapq': <module 'heapq' from
    '/usr/lib/python3.2/heapq.py'>,
'sre_compile': <module 'sre_compile' from '/usr/lib/python3.2/sre_compile.py'>,
...
```

10.1.2 Gestionnaire de contexte (ou bloc gardé)

Utiliser une ressource dans un bloc de code puis terminer par un appel spécifique pour en fermer proprement l'accès (que l'on sorte de ce bloc normalement ou suite à une exception) est un motif récurrent. L'instruction `with` gère élégamment ce problème.

Elle permet [1] à un objet de mettre en place un contexte de « bloc gardé », en assurant l'appel à une méthode dans tous les cas de sortie du bloc.

Cette syntaxe simplifie le code en assurant que certaines opérations sont exécutées avant et après un bloc d'instructions donné. Illustrons ce mécanisme sur un exemple classique où il importe de fermer le fichier utilisé :

```
# au lieu de ce code :
fh = open(filename)
try:
    for line in fh:
        process(line)
finally:
    fh.close()

# il est plus simple d'écrire :
with open(filename) as fh:
    for line in fh:
        process(line)
```

1. https://www.python.org/dev/peps/pep-0343/

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.1.3 Utiliser un dictionnaire pour lancer des fonctions ou des méthodes

L'idée est d'exécuter différentes parties de code en fonction de la valeur d'une variable de contrôle. Certains langages fournissent des instructions `switch` / `case` pour cela. En Python, l'utilisation d'un dictionnaire dans lequel les valeurs stockées sont des fonctions et les clés sont les valeurs de contrôle permet l'activation rapide de la fonction adéquate.

```
animaux = []
nombre_de_felins = 0

def gererChat():
    global nombre_de_felins
    print("Miaou")
    animaux.append("félin")
    nombre_de_felins += 1

def gererChien():
    print("Ouah")
    animaux.append("canidé")

def gererOurs():
    print("Grrr !")
    animaux.append("plantigrade")

dico = {"chat" : gererChat, "chien" : gererChien, "ours" : gererOurs}
betes = ["chat", "ours", "chat", "chien"]  # une liste d'animaux rencontrés

for bete in betes:
    dico[bete]()                           # appel de la fonction correspondante

print("Nous avons rencontré {} félin(s)".format(nombre_de_felins))
print("Familles rencontrées : {}".format(', '.join(animaux), end=" "))
```

L'exécution du script produit :

```
Miaou
Grrr !
Miaou
Ouah
Nous avons rencontré 2 félin(s)
Familles rencontrées : félin, plantigrade, félin, canidé
```

On peut se servir de cette technique par exemple pour implémenter un menu textuel en faisant correspondre des commandes (par exemple une touche au clavier) avec des fonctions à appeler.

10.1.4 Les fonctions récursives

Définition

✍ Une **fonction récursive** comporte un appel à elle-même.

Plus précisément, une fonction récursive doit respecter les deux propriétés suivantes :

1. Une fonction récursive contient un cas de base qui ne nécessite pas de récursion (ce qui évite les récursions sans fin, comme il existe des boucles sans fin).

Copyright © 2017 Dunod.

2. Les appels internes au sein de la fonction doivent s'appliquer sur un problème plus « petit » que le problème traité par l'exécution courante pour se ramener, au final, au cas de base.

Par exemple, trier un tableau de N éléments par ordre croissant, c'est extraire le plus petit élément puis, s'il reste des éléments, trier le tableau restant à $N-1$ éléments.

Un algorithme classique très utile est la méthode de Horner, qui permet d'évaluer efficacement un polynôme de degré n en une valeur donnée x_0, en remarquant que cette réécriture ne contient plus que `n` multiplications :

$$p(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0$$

Voici une implémentation récursive de l'algorithme de Horner dans laquelle le polynôme p est représenté par la liste de ses coefficients $[a_0, \cdots, a_n]$:

```
>>> def horner(p, x):
...     if len(p) == 1:
...         return p[0]
...     p[-2] += x * p[-1]
...     return horner(p[:-1], x)
...
>>> horner([5, 0, 2, 1], 2)  # x**3 + 2*x**2 + 5, en x = 2
21
```

Les fonctions récursives sont souvent utilisées pour traiter les structures arborescentes comme les systèmes de fichiers des disques durs.

Voici l'exemple d'une fonction qui affiche récursivement les fichiers d'une arborescence à partir d'un répertoire fourni en paramètre [1] :

```
from os import listdir
from os.path import isdir, join

def listeFichiersPython(repertoire):
    """Affiche récursivement les fichiers Python à partir de <repertoire>."""
    noms = listdir(repertoire)
    for nom in noms:
        if nom in (".", ".."):  # exclusion répertoire courant et répertoire parent
            continue
        nom_complet = join(repertoire, nom)
        if isdir(nom_complet):  # condition récursive
            listeFichiersPython(nom_complet)
        elif nom.endswith(".py") or nom.endswith(".pyw"):  # condition terminale
            print("Fichier Python :", nom_complet)

listeFichiersPython("/home/bob/Tmp")
```

Dans cette définition, on commence par constituer dans la variable `noms` la liste des fichiers et répertoires du répertoire donné en paramètre. Puis, dans une boucle `for`, tant que l'élément examiné est un répertoire, on rappelle récursivement la fonction sur cet élément pour descendre dans l'arborescence de fichiers. La *condition terminale* est constituée par le `elif` appliqué aux fichiers normaux, qui ajoute un filtrage pour ne lister que les fichiers qui nous intéressent.

Le cas particulier en début de boucle `if nom in ("."`, `".."):` permet de ne pas traiter les répertoires spéciaux que sont le répertoire courant et le répertoire parent.

Le résultat produit est :

1. La fonction standard `os.walk()` fournit ce service de parcours d'arborescence de fichiers.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
Fichier Python : /home/bob/Tmp/parfait_chanceux.py
Fichier Python : /home/bob/Tmp/recursif.py
Fichier Python : /home/bob/Tmp/parfait_chanceux_m.py
Fichier Python : /home/bob/Tmp/verif_m.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone_IHM.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone.py
Fichier Python : /home/bob/Tmp/Truc/calculate.py
Fichier Python : /home/bob/Tmp/Truc/tk_variable.py
```

La récursivité terminale

Définition

✍ On dit qu'une fonction `f` est **récursive terminale**, si tout appel récursif est de la forme :
`return f(…)`
On parle alors d'*appel terminal*.

Python permet la récursivité mais n'optimise pas automatiquement les appels terminaux. Il est donc possible [1] d'atteindre la limite arbitraire fixée à 1 000 appels [2].

On peut pallier cet inconvénient de deux façons. Nous allons illustrer cette stratégie sur un exemple classique, la factorielle.

La première écriture est celle qui découle directement de la définition de la fonction :

```
def factorielle(n):
    """Version récursive non terminale."""
    if n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

On remarque immédiatement (`return n * factorielle(n-1)`) qu'il s'agit d'une fonction récursive **non terminale** car une opération supplémentaire de multiplication doit être réalisée sur le résultat retourné par l'appel récursif. Or une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminal pour la bonne raison qu'il n'y a qu'une phase de descente et pas de phase de remontée.

La méthode classique pour transformer cette fonction en un appel récursif terminal est d'ajouter un argument d'appel jouant le rôle d'accumulateur et permettant de réaliser la multiplication lors de l'appel récursif. D'où le code :

```
def factorielleTerm(n, accu=1):
    """Version récursive terminale."""
    if n == 0:
        return accu
    else:
        return factorielleTerm(n-1, n*accu)
```

La seconde stratégie est d'essayer de transformer l'écriture récursive de la fonction par une écriture itérative. La théorie de la calculabilité montre qu'une telle transformation est toujours possible à partir d'une fonction récursive terminale, ce qu'on appelle l'opération de *dérécursivation*. D'où le code :

1. Voire *incontournable* si on en croit la loi de Murphy…
2. Les fonctions `setrecursionlimit()` et `getrecursionlimit()` du module `sys` permettent de modifier cette limite.

Copyright © 2017 Dunod.

```
def factorielleDerec(n):
    """Version dérécursivée."""
    accu = 1
    while n > 0:
        accu *= n
        n -= 1
    return accu
```

10.1.5 Les listes définies en compréhension

Les listes définies « en compréhension » (souvent appelées « compréhension de listes », expression pas très heureuse calquée sur l'anglais...) permettent de générer ou de modifier des collections de données par une écriture lisible, simple et performante.

Cette construction syntaxique se rapproche de la notation utilisée en mathématiques :
$\{x^2 | x \in [2, 11[\}$ ⇔ `[x**2 for x in range(2, 11)]` ⇒ `[4, 9, 16, 25, 36, 49, 64, 81, 100]`

Définition

✍ Une **liste en compréhension** est une expression littérale de liste équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en compréhension sont utilisables sous trois formes.

Première forme, expression d'une liste simple de valeurs :

```
result1 = [x+1 for x in une_seq]
# a le même effet que :
result2 = []
for x in une_seq:
    result2.append(x+1)
```

Deuxième forme, expression d'une liste de valeurs avec filtrage :

```
result3 = [x+1 for x in une_seq if x > 23]
# a le même effet que :
result4 = []
for x in une_seq:
    if x > 23:
        result4.append(x+1)
```

Troisième forme [1], expression d'une combinaison de listes de valeurs :

```
result5 = [x+y for x in une_seq for y in une_autre]
# a le même effet que :
result6 = []
for x in une_seq:
    for y in une_autre:
        result6.append(x+y)
```

1. Nous limitons nos exemples sur cette troisième forme, mais il est possible d'utiliser plusieurs niveaux de boucles et plusieurs filtrages dans la même liste en compréhension.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Exemples d'utilisations très *pythoniques* :

```
valeurs_s = ["12", "78", "671"]
# conversion d'une liste de chaînes en liste d'entiers
valeurs_i = [int(i) for i in valeurs_s]     # [12, 78, 671]

# calcul de la somme de la liste avec la fonction intégrée sum
print(sum([int(i) for i in valeurs_s]))     # 761

# a le même effet que :
s = 0
for i in valeurs_s:
    s = s + int(i)
print(s)                                    # 761

# Initialisation d'une liste 2D
multi_liste = [[0]*2 for ligne in range(3)]
print(multi_liste)      # [[0, 0], [0, 0], [0, 0]]
```

10.1.6 Les dictionnaires définis en compréhension

Comme pour les listes, on peut définir des dictionnaires en compréhension :

```
>>> {n : x**2 for n, x in enumerate(range(5))}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Notons l'utilisation des accolades **et** du caractère « deux-points », qui sont caractéristiques de la syntaxe des dictionnaires.

10.1.7 Les ensembles définis en compréhension

De même, on peut définir des ensembles en compréhension :

```
>>> {n for n in range(5)}
set([0, 1, 2, 3, 4])
```

Dans ce cas les accolades sont caractéristiques de la syntaxe des ensembles.

10.1.8 Les générateurs et les expressions génératrices

Les générateurs

Définition

✍ Un **générateur** est une fonction [1] qui mémorise son état au moment de produire une valeur. La transmission d'une valeur produite s'effectue en utilisant le mot clé `yield`.

Les générateurs fournissent un moyen de générer des *exécutions paresseuses* [2], ce qui signifie qu'elles ne calculent que les valeurs réellement demandées au fur et à mesure qu'il y en a besoin. Ceci peut s'avérer beaucoup plus efficace (en termes de mémoire) que le calcul, par exemple, d'une énorme liste en une seule fois.

1. Ou plutôt une *procédure* car un générateur ne peut retourner que la valeur `None`.
2. Appelées aussi *appels par nécessité* ou *évaluations retardées*.

Copyright © 2017 Dunod.

Techniquement, un générateur fonctionne en deux temps. D'abord, au lieu de retourner une valeur avec le mot clé `return`, la fonction qui doit servir de générateur utilise le mot clé `yield` pour produire une valeur et se mettre en pause.

Ensuite, à l'utilisation du générateur, le corps de la fonction est exécuté lors des appels implicites dans une boucle `for` (ou bien explicitement en créant d'abord un générateur avec un premier appel à la fonction, puis en utilisant la fonction `next()` sur ce générateur pour produire les valeurs, jusqu'à une exception `StopIteration`).

Voici un exemple de compteur d'entiers qui décrémente l'argument du générateur jusqu'à zéro :

```
>>> def countDown(n):
...     """génère un décompteur à partir de <n>.
...
...     Un générateur ne peut retourner que None.
...     """
...     print('Mise à feu :')
...     while n > 0:
...         yield n
...         n = n - 1

... for val in countDown(5):
...     print(val, end=" ")
...
Mise à feu :
5 4 3 2 1
```

Remarquons que le premier appel au générateur produit trois effets :

1. Création de l'objet générateur par l'appel à la fonction `countDown()`.
2. Initialisation : la fonction `countDown()` se déroule séquentiellement (notons l'affichage `Mise à feu`).
3. Arrivée à l'instruction `yield n`, la fonction retourne la valeur de `n` puis se met en pause.

Les appels suivants déclenchent la reprise de l'exécution de la fonction jusqu'au prochain appel de l'instruction `yield n`. Le mécanisme itère jusqu'au retour de la fonction quand `n` vaut 0.

Les expressions génératrices

Syntaxe

✎ Une expression génératrice possède une syntaxe presque identique à celle des listes en compréhension à la différence qu'une expression génératrice est entourée de parenthèses.

Les expressions génératrices (souvent appelée « genexp ») sont aux générateurs ce que les listes en compréhension sont aux fonctions. Bien qu'il soit transparent, le mécanisme du `yield` vu ci-dessus est encore en action.

Par exemple, la liste en compréhension `for i in [x**2 for x in range(1000000)]:` génère la création d'un million de valeurs en mémoire *avant* de commencer la boucle.

En revanche, dans l'expression `for i in (x**2 for x in range(1000000)):`, la boucle commence *immédiatement* et les valeurs ne sont générées qu'au fur et à mesure des demandes.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.1.9 Fonctions incluses et fermetures

La syntaxe de définition des fonctions en Python permet tout à fait d'*emboîter* leur définition. Voici une fonction incluse simple :

```
def print_msg(msg):
    """Fonction externe."""
    def printer():
        """Fonction incluse."""
        print(msg)
    printer()  # appel à la fonction incluse

print_msg('Hello')  # Hello
```

Le subtil changement suivant définit une fermeture [1] :

```
def print_msg(msg):
    """Fonction externe."""
    def printer():
        """Fonction incluse."""
        print(msg)
    return printer  # retourne la fonction incluse (sans parenthèses : objet fonction)

fct = print_msg('Hello')  # fct est une fonction
fct()  # Hello
```

Définition

Une **fermeture** réunit ces trois critères :

1. C'est une fonction qui doit comporter une fonction incluse.
2. La fonction incluse doit utiliser une valeur définie dans la fonction externe, qu'elle mémorise (on parle de « capture de contexte ») lors de sa définition.
3. La fonction externe doit retourner la fonction incluse.

Les fermetures évitent l'utilisation des variables globales. Elles permettent d'attacher un état à une fonction, tout comme la programmation objet permet d'encapsuler un état dans un objet. Quand une classe comporte peu de méthodes, la fermeture est une alternative élégante.

Fonction fabrique

Le besoin est de créer des instances de fonctions ou de classes suivant certaines conditions. Un bon moyen est d'implémenter la création d'un objet souple en utilisant une fonction *fabrique*.

Idiome de la fonction fabrique [2] renvoyant une fermeture :

```
def creer_plus(ajout):
    """Fonction 'fabrique'."""
    def plus(x):
        """Fonction 'fermeture' : utilise des noms locaux à creer_plus()."""
        return ajout + x
    return plus
```

1. En anglais *closure*.
2. En anglais *factory*.

Copyright © 2017 Dunod.

```
p = creer_plus(23)
q = creer_plus(42)
print("p(100) =", p(100))  # ('p(100) =', 123)
print("q(100) =", q(100))  # ('q(100) =', 142)
```

Fonction fabrique renvoyant une classe :

```
class Fixe:
    def allumer(self):
        print("Appuyer sur interrupteur en façade du boîtier.")

class Portable:
    def allumer(self):
        print("Ouvrir écran, appuyer sur bouton ON/OFF au bas de l'écran.")

def ordinateur(mobile=False):
    if mobile:
        return Portable()
    else:
        return Fixe()

mon_pc = ordinateur()  # appel au Fixe
mon_pc.allumer()       # 'Appuyer sur interrupteur en façade du boîtier.'
```

10.1.10 Les décorateurs

Les décorateurs permettent d'encapsuler la définition d'une fonction (ou méthode ou classe) et de transformer le résultat de cette définition. Cela permet par exemple d'ajouter des *prétraitements* ou des *post-traitements* lors de l'appel d'une fonction ou d'une méthode.

Le décorateur lui-même est simplement défini comme une fonction, prenant au moins comme paramètre l'objet à décorer. Il doit retourner l'objet qu'il a décoré ou bien un moyen d'accès transparent à cet objet (on parle souvent de *wrapper*, terme anglais pour « emballage »).

Il est appliqué à une définition (de fonction ou méthode ou classe) simplement en utilisant la notation `@`, suivie du nom du décorateur, immédiatement avant la définition à traiter.

Syntaxe

✎ Soit `deco()` un décorateur défini ainsi :

```
def deco(uneFct):
    print("Décoration de", uneFct)  # par exemple
    return uneFct
```

Pour « décorer » une fonction à l'aide de ce décorateur, on écrit simplement :

```
@deco
def fonction(arg1, arg2, ...):
    pass
```

© Dunod. Toute reproduction non autorisée est un délit.
Copyright © 2017 Dunod.

Une fonction peut être multi-décorée :

```
def decor1():
    ...

def decor2():
    ...

def decor3():
    ...

@decor1 @decor2 @decor3
def g():
    pass
```

Ceci correspond à une définition de g :

```
def g() :
    pass

g = decor1(decor2(decor3(g)))
```

Voici un exemple simple :

```
def unDecorateur(f):
    cptr = 0
    def _interne(*args, **kwargs):
        nonlocal cptr
        cptr = cptr + 1
        print("Fonction décorée :", f.__name__, ". Appel numéro :", cptr)
        return f(*args, **kwargs)

    return _interne

@unDecorateur
def uneFonction(a, b):
    return a + b

def autreFonction(a, b):
    return a + b

# programme principal ================================================
print(uneFonction(1, 2))                    # utilisation d'un décorateur
autreFonction = unDecorateur(autreFonction)  # utilisation de la composition de fonction
print(autreFonction(1, 2))

print(uneFonction(3, 4))
print(autreFonction(6, 7))
```

Ce qui affiche :

```
Fonction décorée : uneFonction. Appel numéro : 1
3
Fonction décorée : autreFonction. Appel numéro : 1
3
Fonction décorée : uneFonction. Appel numéro : 2
7
```

Copyright © 2017 Dunod.

```
Fonction décorée : autreFonction. Appel numéro : 2
13
```

Note : Le module `functools` fournit une fonction `update_wrapper()` et un décorateur `wraps()` permettant de reproduire les caractéristiques de la fonction de base dans la fonction wrappée (docstring, paramètres par défaut, etc.). Ceci permet un bon fonctionnement des outils basés sur l'introspection avec les fonctions ainsi emballées dans des wrappers.

10.2 Techniques objets

Comme nous l'avons vu lors du chapitre précédent, Python est un langage complètement objet. Tous les types de base ou dérivés sont en réalité des types de données implémentés sous forme de classe.

10.2.1 Les *Functors*

En Python, un objet fonction ou *functor* est une référence à tout objet « appelable » [1] : fonction, fonction anonyme lambda [2], méthode, classe. La fonction prédéfinie `callable()` permet de tester cette propriété :

```
>>> def maFonction():
...     print('Ceci est "appelable"')
...
>>> callable(maFonction)
True
>>> chaine = 'Une chaîne'
>>> callable(chaine)
False
```

Il est possible de transformer les instances d'une classe en *functor* si la méthode spéciale `__call__` est définie dans la classe :

```
>>> class A:
...     def __init__(self):
...         self.historique = []
...     def __call__(self, a, b):
...         self.historique.append((a, b))
...         return a + b
...
>>> a = A()
>>> a(1, 2)
3
>>> a(3, 4)
7
>>> a(5, 6)
11
>>> a.historique
[(1, 2), (3, 4), (5, 6)]
```

1. *Callable* en anglais.
2. Cette notion sera développée ultérieurement [cf. p. 137]

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.2.2 Les accesseurs

Le problème de l'encapsulation

Dans le paradigme objet, la *visibilité* de l'attribut d'un objet est **privée**, les autres objets n'ont pas le droit de le consulter ou de le modifier.

En Python, tous les attributs d'un objet sont de visibilité publique, donc accessibles depuis n'importe quel autre objet. On peut néanmoins remédier à cet état de fait.

Lorsqu'un nom d'attribut est préfixé par un caractère souligné, il est conventionnellement réservé à un usage interne (privé). Mais Python n'oblige à rien [1], c'est au développeur de respecter la convention !

On peut également préfixer un nom par deux caractères « souligné », ce qui permet d'éviter les collisions de noms dans le cas où un même attribut serait défini dans une sous-classe. Ce renommage [2] a comme effet de bord de rendre l'accès à cet attribut plus difficile de l'extérieur de la classe qui le définit, quoique cette protection reste déclarative et n'offre pas une sécurité absolue.

Enfin, l'état de l'attribut d'un objet peut être géré par des accesseurs (ou simplement méthodes d'accès). On distingue habituellement le *getter* pour la lecture, le *setter* pour la modification et le *deleter* pour la suppression.

La solution `property`

Le principe de l'encapsulation est mis en œuvre par la notion de propriété.

Définition

✍ Une **propriété** (`property`) est un attribut d'instance possédant des fonctionnalités spéciales.

Les *property* utilisent la syntaxe des décorateurs. Bien remarquer que, dans l'exemple suivant, on utilise `artist` et `title` comme des attributs simples :

```
class Oeuvre:

    def __init__(self, artist, title):
        self.__artist = artist
        self.__title = title

    @property
    def artist(self):
        return self.__artist

    @artist.setter
    def artist(self, artist):
        self.__artist = artist

    @property
    def title(self):
        return self.__title

    @title.setter
    def title(self, title):
        self.__title = title
```

1. Slogan des développeurs Python : *We're all consenting adults here* (« nous sommes entre adultes consentants »).
2. Le nom est préfixé de façon interne par `_NomClasse`.

Copyright © 2017 Dunod.

```
    def __str__(self):
        return "{:s} : '{:s}' de {:s}".format(self.__class__.__name__, self.__title, self.__artist)

if __name__ == '__main__':
    items = []

    items.append(Oeuvre('François Rabelais', 'Gargantua'))
    items.append(Oeuvre('Charles Baudelaire', 'Les Fleurs du mal'))

    for item in items:
        print("{} : '{}'".format(item.artist, item.title))
```

Ce qui produit l'affichage :

```
François Rabelais : 'Gargantua'
Charles Baudelaire : 'Les Fleurs du mal'
```

Un autre exemple : la classe `Cercle`

Schéma de conception : nous allons tout d'abord définir une classe `Point` que nous utiliserons comme classe de base de la classe `Cercle`, en considérant qu'un cercle est un point (son centre) de grande dimension (avec un rayon).

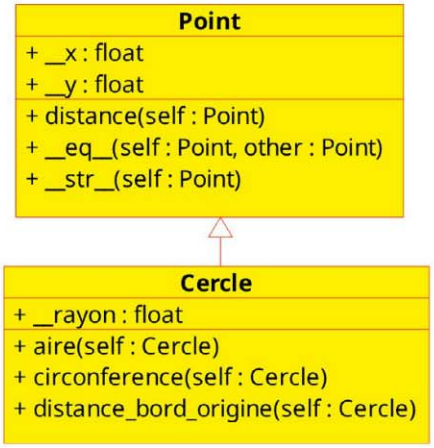


FIGURE 10.1 – Conception UML de la classe `Cercle`

Voici le code de la classe `Point` :

```
class Point:

    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y

    @property
    def distance_origine(self):
        return math.hypot(self.__x, self.__y)

    def __eq__(self, other):
```

© Dunod. Toute reproduction non autorisée est un délit.
Copyright © 2017 Dunod.

```
        return self.__x == other.__x and self.__y == other.__y

    def __str__(self):
        return "({}, {})".format(self.__x, self.__y)
```

L'emploi de la solution `property` permet un accès en *lecture seule* au résultat de la méthode `distance_origine` considérée alors comme un simple attribut (car on l'utilise sans parenthèses). Cet accès se fait en lecture seule car le *setter* correspondant n'a pas été défini :

```
p1, p2 = Point(), Point(3, 4)
print(p1 == p2)                    # False
print(p2, p2.distance_origine)     # (3, 4) 5.0
```

De nouveau, les méthodes renvoyant un simple flottant seront utilisées comme des attributs en lecture seule grâce à l'utilisation de `property` :

```
class Cercle(Point):
    def __init__(self, rayon, x=0, y=0):
        super().__init__(x, y)
        self.__rayon = rayon

    @property
    def aire(self): return math.pi * (self.__rayon ** 2)

    @property
    def circonference(self): return 2 * math.pi * self.__rayon

    @property
    def distance_bord_origine(self):
        return abs(self.distance_origine - self.__rayon)
```

Voici la syntaxe permettant d'utiliser la méthode rayon comme un attribut en *lecture-écriture*. Remarquez que la méthode `rayon()` retourne l'attribut protégé : `__rayon` qui sera modifié par le *setter* (la méthode modificatrice) :

```
    @property
    def rayon(self):
        return self.__rayon

    @rayon.setter
    def rayon(self, rayon):
        if rayon <= 0 :   # Contrôle de validité de la valeur.
            raise ValueError("Le rayon doit être strictement positif")
        self.__rayon = rayon
```

Exemple d'utilisation des instances de `Cercle` :

```
    def __eq__(self, other):
        return (self.rayon == other.rayon
                            and super().__eq__(other))

    def __str__(self):
        return ("{0.__class__.__name__}({0.rayon!s}, {0.x!s}, "
                "{0.y!s})".format(self))
```

Copyright © 2017 Dunod.

```
if __name__ == "__main__":
    c1 = Cercle(2, 3, 4)
    print(c1, c1.aire, c1.circonference)
    print(c1.distance_bord_origine, c1.rayon)
    c1.rayon = 1                    # modification du rayon
    print(c1.distance_bord_origine, c1.rayon)
```

Ce qui affiche :

```
Cercle(2, 3, 4) 12.5663706144 12.5663706144
3.0 2
4.0 1
```

10.2.3 Le *duck typing*...

Il existe un style de programmation très pythonique appelé *duck typing* :

« S'il marche comme un canard et cancane comme un canard, alors c'est un canard ! »

Cela signifie que Python ne s'intéresse qu'au *comportement* des objets. Si des objets offrent la même API (interface de programmation), l'utilisateur peut employer les mêmes méthodes. C'est une différence majeure par rapport aux langages dits statiquement typés comme C++, Java ou C#, qui nécessitent obligatoirement d'intégrer les classes utilisées dans une hiérarchie fixée.

Prenons l'exemple d'un script qui a besoin d'écrire dans un fichier en utilisant `write(s)`, tout objet qui supporte cette méthode sera accepté. La fonction suivante prend en paramètre un objet fichier texte ouvert et y écrit la représentation d'une table de multiplication.

```
def genere_table_multi(f, n):
    entete = "      "
    for j in range(1, n+1):
        entete += "{: 4d}  ".format(j)
    f.write(entete + '\n')
    f.write("-"*((n+1)*6) + '\n')
    for i in range(1, n+1):
        ligne = "{: 4d} |".format(i)
        for j in range(1, n+1):
            ligne += "{: 4d}  ".format(i*j)
        f.write(ligne + '\n')
```

On l'utiliserait simplement ainsi :

```
with open("tablemulti.txt", "w") as f:
    genere_table_multi(f, 5)
```

Fichier résultat :

```
          1     2     3     4     5
-----------------------------------
   1 |    1     2     3     4     5
   2 |    2     4     6     8    10
   3 |    3     6     9    12    15
   4 |    4     8    12    16    20
   5 |    5    10    15    20    25
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Il est possible de définir une classe qui serve d'emballage (*wrapper*) au fichier et ajoute à chaque écriture une indication d'horodatage. La fonction ne change pas, on lui fournit simplement l'objet qui supporte le *duck typing* de la méthode `write()`.

```
import datetime

class FichierHorodate:
    """Ajout d'une indication de date et heure devant les écritures.

    Amélioration à faire: détecter l'écriture de lignes complètes effectives.
    """
    def __init__(self, fichier):
        self.fichier = fichier

    def write(self, s):
        self.fichier.write(datetime.datetime.now().strftime("%H.%M.%S.%f:"))
        self.fichier.write(s)


with open("tablemultidatee.txt", "w") as f:
    wrapfic = FichierHorodate(f)
    genere_table_multi(wrapfic, 5)
```

On obtient un fichier avec chaque ligne horodatée à la microseconde :

```
 12.52.10.380836:         1     2     3     4     5
 12.52.10.380898:-------------------------------------
 12.52.10.380935:   1 |   1     2     3     4     5
 12.52.10.380966:   2 |   2     4     6     8    10
 12.52.10.380996:   3 |   3     6     9    12    15
 12.52.10.381024:   4 |   4     8    12    16    20
 12.52.10.381051:   5 |   5    10    15    20    25
```

10.2.4 Le *duck typing*... et les annotations

L'aspect dynamique de Python, où la nature des données est dynamiquement découverte à l'exécution, offre l'avantage de pouvoir utiliser le *duck typing*, avec un code court et clair.

Mais, quand le programme reçoit un type inattendu, il s'arrête en erreur à l'exécution. C'est bien là la différence avec les langages à typage statique, pour lesquels toute erreur de type est décelée en amont dès la phase de compilation. Les *annotations* ont été pensées pour pallier ce problème.

Syntaxe

Exemple d'annotation :

```
def pgcd(a:int, b:int) -> int:
    while b:
        a,b = b, a%b
    return a
```

Les annotations permettent notamment de fournir des informations supplémentaires associées aux fonctions ou méthodes, pouvant spécifier par exemple les types attendus et retournés. Or, c'est

Copyright © 2017 Dunod.

important, ces informations *optionnelles* n'ont aucun impact sur l'exécution du code, elles sont simplement stockées comme attributs lors de la compilation par l'interpréteur Python. Des outils tierces parties [1] pourront les utiliser pour par exemple :

- faire de la vérification statique de type utile dans certains cas (gros projets, nombreux développeurs, aide à la documentation et au débogage complexe...) ;
- fournir une aide aux éditeurs de code ;
- offrir un complément à la documentation des *docstrings* ;
- ...

10.3 Techniques fonctionnelles

10.3.1 Directive `lambda`

Issue de langages fonctionnels (comme OCaml, Haskell, Lisp), la directive `lambda` permet de définir un objet *fonction anonyme* comportant un bloc d'instructions limité à une expression dont l'évaluation fournit la valeur de retour de la fonction.

Ces fonctions anonymes sont souvent utilisées lorsqu'il s'agit simplement d'adapter l'appel à une fonction existante, par exemple dans les `callbacks` des interfaces graphiques... Nous avons utilisé une fonction lambda dans la classe `Allo_IHM` de l'exemple du chapitre 9.3.2 :

```
        self.listeSelection.bind("<Double-Button-1>", lambda event: self.cb_afficher())
```

Syntaxe

`lambda` *[paramètres]*`:` `expression`

Par exemple, cette fonction retourne « s » si son argument est différent de 1, une chaîne vide sinon :

```
>>> s = lambda x: "" if x == 1 else "s"
>>> s(3)
's'
>>> s(1)
''
```

Associées aux fermetures, les fonctions anonymes permettent de créer simplement des fonctions de calcul paramétrées :

```
def polynome(a, b, c) :
    return lambda x : a*x**2 + b*x + c

p1 = polynome(3, -1, 4)
p2 = polynome(-1, 2, 0)
print(p1(1))   # 6
print(p1(2))   # 14
print(p2(10))  # -80
```

1. En particulier le projet mypy, auquel GvR, le créateur de Python, participe activement.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.3.2 Les fonctions `map`, `filter` et `reduce`

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative, qui met en avant les changements d'état [1]. Elle repose sur trois concepts : *mapping* (correspondance), *filtering* (filtrage) et *reducing* (réduction), qui sont implémentés en Python par trois fonctions : `map()`, `filter()` et `reduce()`.

La fonction `map()` :

`map(fonction, séquence)` construit et renvoie un générateur dont les valeurs produites sont les résultats de l'application de la fonction aux valeurs de la séquence :

```
>>> map(lambda x:x**2, range(10))
<map object at 0x7f3a80104f50>
>>> list(map(lambda x:x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On remarque que `map()` peut être remplacée par un générateur en compréhension.
Pour notre exemple : `(x**2 for x in range(10))`

La fonction `filter()` :

`filter(fonction, séquence)` construit et renvoie un générateur dont les valeurs produites sont celles pour lesquelles l'application de la fonction aux valeurs de la séquence a retourné vrai :

```
>>> list(filter(lambda x: x > 4, range(10)))
[5, 6, 7, 8, 9]
```

De même, `filter()` peut être remplacée par un générateur en compréhension.
Dans notre exemple : `(x for x in range(10) if x > 4)`

La fonction `reduce()` :

`reduce()` est une fonction du module `functools`. Elle applique de façon cumulative une fonction de deux arguments aux éléments d'une séquence, de gauche à droite, de façon à réduire cette séquence à une seule valeur qu'elle renvoie.

Un petit exemple pour montrer son fonctionnement :

```
from  functools import reduce

def somme(x, y):
    print(x,'+',y,'=>',x+y)
    return x + y

reduce(somme, [1, 2, 3, 4, 5])
```

Produit :

```
1 + 2 => 3
3 + 3 => 6
6 + 4 => 10
10 + 5 => 15
```

La fonction `reduce()` peut, dans certains cas, être avantageusement remplacée par une des fonctions suivantes : `all()`, `any()`, `max()`, `min()` ou `sum()`. Par exemple :

1. https://fr.wikipedia.org/wiki/Programmation_impérative

Copyright © 2017 Dunod.

```
>>> sum([1, 2, 3, 4, 5])
15
```

Il est aussi possible d'utiliser `reduce` avec le module `operator`, qui fournit les opérateurs Python sous forme de fonctions.

```
>>> reduce(operator.mul, range(10, 101, 10))
3628800000000000
```

10.3.3 Les applications partielles de fonctions

Issue de la programmation fonctionnelle, une PFA (application partielle de fonction) de n paramètres prend le premier argument comme paramètre fixe et retourne un objet fonction (ou instance) utilisant les $n-1$ arguments restants. En Python la définition d'une fonction PFA permet de spécifier plusieurs des premiers paramètres positionnels, ainsi que des paramètres nommés.

Les PFA sont utiles dans les fonctions de calcul comportant de nombreux paramètres. On peut en fixer certains et ne faire varier que ceux sur lesquels on veut agir :

```
from functools import partial
def f(m, c, d, u):
    return 1000*m + 100*c + 10*d + u
print(f(1, 2, 3, 4))      # 1234
g = partial(f, 1, 2, 3)
print(g(4), g(0))         # (1234, 1230)
h = partial(f, 1, 2)
print(h(3, 4), h(0, 1))   # (1234, 1201)
```

Les PFA sont aussi utiles dans le cadre de la programmation d'interfaces graphiques, pour fournir des modèles partiels de widgets préconfigurés (ceux-ci ont souvent de nombreux paramètres).

10.3.4 Programmation fonctionnelle *pure*

Nous avons déjà pu voir, lors de la présentation de la portée des objets [cf. p. 66], qu'il est possible de définir des variables globales qui existent avant, pendant et après l'appel de fonctions. Et, dans la présentation des arguments mutables [cf. p. 65], nous avons vu qu'il était possible d'effectuer des modifications de données passées en paramètre qui persistent après la fin de la fonction.

En programmation fonctionnelle, une fonction est dite *pure* (ou *propre*) lorsqu'elle n'a pas d'effet de bord [cf. p. 65] et que son résultat dépend uniquement des paramètres en entrée (donc pas d'une information qui serait lue au cours de l'exécution de la fonction). Elle peut produire une valeur mais interne à la fonction et qui n'est retournée au programme que comme valeur de retour de la fonction. Une telle fonction est beaucoup plus facile à vérifier et à maintenir, et sa réutilisation est facilitée.

Prenons l'exemple simplifié d'une fonction qui recherche la liste des communes ayant un nom proche d'un nom saisi, afin de pouvoir choisir une commune spécifique — le genre d'algorithme que l'on a typiquement sur des pages web lorsqu'on saisit certains lieux.

Pour retourner la valeur, la première version de la fonction utilise un effet de bord en remplissant une liste passée en paramètre :

```
COMMUNES = { 75001: "Paris" }   # Mapping code: nom des 36000 communes

def recherche_communes_proches(nom, lst):
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
    for code, nomv in COMMUNES.items():
        if proche(nom, nomv):  # algo à votre …choix
            lst.append(code)
# Appel:
lstcom = []
recherche_communes_proches("Paris", lstcom)
```

C'est l'appelant qui fournit la liste à remplir (elle doit donc exister avant l'exécution de la fonction). S'il oublie de vider la liste entre les appels, les résultats vont s'accumuler dedans (la fonction pourrait faire un `lst.clear()` avant de faire sa boucle pour éviter ce problème).

Première amélioration, on construit la valeur résultat complètement dans la fonction, et on la retourne à la fin. Le code devient alors *fonctionnel* :

```
COMMUNES = { 75001: "Paris" }   # Mapping code: nom des 36000 communes

def recherche_communes_proches(nom):
    lst = []
    for code, nomv in COMMUNES.items():
        if proche(nom, nomv):  # algo à votre …choix
            lst.append(code)
    return lst
# Appel:
lstcom = recherche_communes_proches("Paris")
```

Cette fonction ne modifie pas son environnement, mais elle se réfère à une variable globale, ce qui en limite l'usage. On va procéder à une seconde amélioration afin de la rendre *pure* :

```
COMMUNES = { 75001: "Paris" }   # Mapping code: nom des 36000 communes

def recherche_communes_proches(nom, lieux=COMMUNES):
    lst = []
    for code, nomv in lieux.items():
        if proche(nom, nomv):  # algo à votre …choix
            lst.append(code)
    return lst
# Appel:
lstcom = recherche_communes_proches("Paris")
```

On peut maintenant facilement la tester en utilisant comme `lieux` des dictionnaires contenant les valeurs sur lesquelles on veut vérifier l'algorithme de `proche()`. Et le code est devenu suffisamment générique pour être potentiellement utilisable dans d'autres cas, il suffit de fournir un paramètre pour `lieux` qui remplacera la variable globale utilisée par défaut.

Une dernière étape d'amélioration, dans laquelle le code générique de l'algorithme est nommé avec du sens et dans laquelle le code spécifique à notre cas d'usage est identifié (sans que le reste du programme ne soit modifié) :

```
def recherche_noms_proches(nom, codesnoms):
    lst = []
    for code, nomv in codesnoms.items():
        if proche(nom, nomv):  # algo à votre …choix
            lst.append(code)
    return lst
COMMUNES = { 75001: "Paris" }   # Mapping code: nom des 36000 communes

def recherche_communes_proches(nom):
```

Copyright © 2017 Dunod.

```
    return recherche_noms_proches(nom, COMMUNES)
# Appel:
lstcom = recherche_communes_proches("Paris")
```

10.4 La persistance et la sérialisation

Définition

✍ La **persistance** consiste à sauvegarder des données afin qu'elles survivent à l'arrêt de l'application.

On peut distinguer deux étapes :

— la sérialisation et la désérialisation ;

— le stockage et le rapatriement.

La **sérialisation** est le processus de conversion d'un ensemble d'objets en un flux d'octets ; celui-ci peut ensuite être enregistré sur disque, transmis par réseau, enregistré dans une base de données, etc. Le format du flot d'octets peut être du texte lisible avec une syntaxe décrivant un format structuré, ou bien un codage binaire dédié avec un format nécessitant obligatoirement des outils spécifiques pour être lu par un humain.

Inversement, la **désérialisation** recrée les données d'origine à partir du flux d'octets.

Examinons des exemples simples.

10.4.1 Sérialisation avec `pickle` et `json`

Le module `pickle`

L'intérêt du module `pickle` est sa simplicité. Par contre, ce n'est pas un format utilisable avec d'autres langages, il faut le réserver à des cas où l'on peut rester uniquement dans le monde Python.

La sérialisation avec `pickle` produit un tableau d'octets (type Python `bytes`). On l'utilise généralement avec un fichier [1] ouvert en mode binaire (contrairement au mode texte, que l'on a déjà vu cf. § 5.1 p. 53), avec le mode `"wb"`. Par exemple pour un dictionnaire :

```
import pickle

favorite_color = {"lion": "jaune", "fourmi": "noire", "caméléon": "variable"}
# stocke ses données dans un fichier
with open("pickle_tst", "wb") as f:
    pickle.dump(favorite_color, f)

# retrouver ses données : pickle recrée un dictionnaire
with open("pickle_tst", "rb") as f:
    dico = pickle.load(f)
print(dico)
```

L'affichage des données relues produit :

```
{'fourmi': 'noire', 'lion': 'jaune', 'caméléon': 'variable'}
```

1. Si l'on veut récupérer directement en mémoire le flux d'octets, on peut utiliser un pseudo-fichier de la classe io.BytesIO, qui capturera ce flux.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

`Pickle` est utilisable afin de sérialiser ses propres classes. Si l'introspection ne permet pas au module de sérialiser les attributs, il faut alors définir des méthodes supplémentaires pour permettre d'extraire et de restaurer un état de l'objet.

Le module `json`

Le module `json` permet d'encoder et de décoder des informations au format `json`[1]. C'est un format d'échange très utile, implémenté dans un grand nombre de langages pour échanger des données structurées d'une façon standardisée. La représentation des données est un texte lisible et se rapproche d'ailleurs beaucoup de la syntaxe Python. Les types de base (numériques, chaînes, booléens, conteneurs liste ou dictionnaire...) sont supportés directement, par contre il faudra écrire des fonctions d'aide à la sérialisation pour supporter d'autres types de données.

On utilise la même syntaxe qu'avec `pickle`, à savoir `dump()` et `load()`, qui permettent de sérialiser vers/depuis un fichier, textuel cette fois. Le module fournit aussi les fonctions `dumps()` et `loads()` pour travailler directement avec des chaînes :

```
import json

# encodage dans un fichier
with open("json_tst", "w") as f:
    json.dump(['foo', {'bar':('baz', None, 1.0, 2)}], f)

# décodage
with open("json_tst") as f:
    print(json.load(f))
```

Le fichier `json_tst` contient :

```
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

10.4.2 Stockage avec `sqlite3`

Le module `sqlite3` est une bibliothèque écrite en C qui implémente une base de données relationnelle légère utilisant des fichiers (voire la mémoire) pour le stockage et le langage standard de requêtes SQL pour les manipulations de données.

Cet outil convient bien à un usage impliquant des volumes de données raisonnables et permet aussi de maquetter une application avant le passage à des bases de données plus avancées (fonctionnement client/serveur multi-utilisateur et capables de monter en puissance).

Nous nous contenterons ici d'un court exemple d'utilisation de `sqlite`, l'apprentissage des bases de SQL nécessiterait au minimum un chapitre dédié... et pour bien faire un ouvrage complet – il en existe déjà de très bien faits (cf. [7], p. 217). On pourra aussi se reporter au site `http://sql.sh/`, qui fournit des cours accessibles.

Il faut noter que cette bibliothèque permet de partager des données avec des applications écrites dans les nombreux langages qui supportent `sqlite3`. Ceci permet par exemple de réaliser des manipulations de données en Python en utilisant un fichier `sqlite3` pour le stockage, puis d'utiliser celui-ci dans un cadre de génération de mailing avec un traitement de texte.

Exemple de stockage d'une table :

1. *JavaScript Object Notation.*

Copyright © 2017 Dunod.

```
import sqlite3

path = '/home/bob/Dunod/3-corps/ch10/src/telBD'
conn = sqlite3.connect(path)     # création du connecteur
with conn as c:                  # création du curseur d'exécution de requêtes
    # création de la table
    c.execute("""create table tel (nom text, prenom text, numero integer)""")
    # insertion de lignes de données
    c.execute("""insert into tel values ('Caillebotte', 'Gustave', '02 13 45 67 89')""")
    c.execute("""insert into tel values ('Dirac', 'Paul', '01 23 45 67 89')""")
    c.execute("""insert into tel values ('Einstein', 'Albert', '03 21 45 67 89')""")
    c.execute("""insert into tel values ('Hélias', 'Pierre Jakez', '04 23 15 67 89')""")
    c.execute("""insert into tel values ('Monet', 'Claude', '05 23 41 67 89')""")
    c.execute("""insert into tel values ('Seurat', 'Georges', '06 23 45 17 89')""")
```

Le fichier `tel_bd` produit peut être visualisé par le programme *SQLite database browser* (☞ Fig. 10.2).

	nom	prenom	numero
	Filter	Filter	Filter
1	Caillebotte	Gustave	02 13 45 67 89
2	Dirac	Paul	01 23 45 67 89
3	Einstein	Albert	03 21 45 67 89
4	Hélias	Pierre Jakez	04 23 15 67 89
5	Monet	Claude	05 23 41 67 89
6	Seurat	Georges	06 23 45 17 89

FIGURE 10.2 – Visualisation d'un fichier de base de données sqlite3

10.5 Les tests

Dès lors qu'un programme dépasse le stade du petit script, le problème des erreurs et donc des tests se pose inévitablement.

Définition

✍ Un **test** consiste à appeler la fonctionnalité spécifiée dans le cahier des charges de l'application, avec un scénario qui correspond à un cas d'utilisation, et à vérifier que cette fonctionnalité se comporte comme prévu.

10.5.1 Tests unitaires et tests fonctionnels

On distingue deux familles de test :

— Tests unitaires : validations isolées du fonctionnement d'une classe, d'une méthode ou d'une fonction.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

— Tests fonctionnels : validation de l'application complète comme une « boîte noire » en la manipulant ainsi que le ferait l'utilisateur final. Ces tests doivent passer par les mêmes interfaces que celles fournies aux utilisateurs, c'est pourquoi ils sont spécifiques à la nature de l'application et plus délicats à mettre en œuvre.

Dans cette introduction, nous nous limiterons à une courte présentation des tests unitaires.

10.5.2 Module `unittest`

Le module standard `unittest` fournit un outil similaire à ce que l'on retrouve dans d'autres langages : `JUnit` (Java), `NUnit` (.Net), `JSUnit` (JavaScript), tous dérivés d'un outil initialement développé pour le langage SmallTalk : `SUnit`.

Par convention, chaque module est associé à un module de tests unitaires, placé dans un répertoire `tests` du paquet. Par exemple, un module nommé `calculs.py` aura un module de tests nommé `tests/test_calculs.py`.

`unittest` propose une classe de base, `TestCase`. Chaque méthode implémentée dans une classe dérivée de `TestCase`, et préfixée de `test_`, sera considérée comme un test unitaire [1] :

```
# fichier calculs.py
"""Module de calculs."""

# fonctions
def moyenne(*args):
    """Renvoie la moyenne."""
    length = len(args)
    sum = 0
    for arg in args:
        sum += arg
    return sum/length

def division(a, b):
    """Renvoie la division."""
    return a/b
```

```
"""Module de test du module de calculs."""

# import ------------------------------------------------------------------
import sys
import unittest
from os.path import abspath, dirname
# on enrichit le path pour ajouter le répertoire absolu du source à tester :
sys.path.insert(0, dirname(dirname((abspath(__file__)))))
from calculs import moyenne, division

# définition de classe et de fonction ------------------------------------
class CalculTest(unittest.TestCase):

    def test_moyenne(self):
        self.assertEquals(moyenne(1, 2, 3), 2)
        self.assertEquals(moyenne(2, 4, 6), 4)
```

1. Cf. [2], p. 217.

Copyright © 2017 Dunod.

```
    def test_division(self):
        self.assertEquals(division(10, 5), 2)
        self.assertRaises(ZeroDivisionError, division, 10, 0)

def test_suite():
    tests = [unittest.makeSuite(CalculTest)]
    return unittest.TestSuite(tests)

# auto-test ======================================================================
if __name__ == '__main__':
    unittest.main()
```

L'exécution du test produit :

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Pour effectuer une « campagne de tests », il reste à créer un script qui :

- recherche tous les modules de test : leurs noms commencent par `test_` et ils sont contenus dans un répertoire **`tests`** ;
- récupère la `suite`, renvoyée par la fonction `test_suite()` de chaque module ;
- crée une suite de suites et lance la campagne.

Note : Des outils comme **`buildbot`** [1] (écrit... en Python) permettent de réaliser ce que l'on appelle de l'« intégration continue », c'est-à-dire l'application automatique de suites de test sur un logiciel en développement afin de vérifier qu'il peut être construit à partir des sources et qu'il fonctionne correctement avec les dernières modifications qui y ont été apportées.

10.6 La documentation des sources

Durant la vie d'un projet, on distingue plusieurs types de documentations :

- les **documents de spécification** (ensemble explicite d'exigences à satisfaire) ;
- les **documents techniques** attachés au code ;
- les **manuels d'utilisation** et autres documents de haut niveau.

Les documents techniques évoluent au rythme du code et peuvent donc être traités comme lui : ils devraient pouvoir être lus et manipulés avec un simple éditeur de texte afin de s'intégrer aux outils de contrôle et de suivi des sources mis en place pour le code lui-même.

Il existe deux outils majeurs pour concevoir des documents pour les applications Python à partir d'un format simple texte :

- un format texte enrichi : **`reStructuredText`** ;
- les **`doctests`**, compatibles avec ce format. Ils permettent de combiner les textes des **`doc-strings`** des modules, classes, fonctions avec les tests à réaliser.

1. http://buildbot.net/

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.6.1 Le format reST

Le format **reStructuredText**, communément appelé **reST**, est un système de balises légères et extensibles utilisées pour ajouter des marques dans des textes.

À la différence de LaTeX ou d'HTML il enrichit le document de manière « non intrusive », c'est-à-dire que les fichiers restent directement lisibles.

docutils

Le projet **docutils**, qui inclut l'interpréteur **reST**, fournit un jeu d'utilitaires :

- **rst2html** génère un rendu HTML avec une feuille de style css intégrée ;
- **rst2latex** crée un fichier LaTeX équivalent ;
- **rst2s5** construit une présentation au format s5, qui permet de créer des présentations interactives en HTML.

Sphinx

Sphinx est un logiciel libre de type générateur de documentation. Il s'appuie sur des fichiers au format reStructuredText, qu'il sait convertir en HTML, PDF, man, et autres formats.

De très nombreux projets utilisent Sphinx pour leur documentation officielle, par exemple MathJax. Cet outil est même utilisé pour produire des livres. Cf. `http://sphinx-doc.org/examples.html`.

rst2pdf

Le programme **rst2pdf** génère directement une documentation au format PDF.

Ci-après est donné un exemple [1] simple de fichier texte au format reST, suivi du résultat produit par son traitement par **rst2pdf**.

On remarque entre autres que :

- la principale balise est la **ligne blanche** qui sépare les différentes structures du texte ;
- la structuration se fait en soulignant les titres des sections de différents niveaux avec des caractères de ponctuation (`= - _ : ,` etc.). À chaque fois qu'il rencontre un texte ainsi souligné, l'interpréteur associe le caractère utilisé à un niveau de section ;
- un titre est généralement souligné et surligné avec le même caractère ;
- l'utilisation de `..` en début de ligne permet de spécifier des blocs de texte ayant des caractéristiques spécifiques.

```
======================
Fichier au format reST
======================

Section 1
=========
On est dans la section 1.

Sous-section
~~~~~~~~~~~~
Ceci est une sous-section.
```

1. Cf. [2], p. 217

Copyright © 2017 Dunod.

```
Sous-sous-section
.................
Ceci est une sous-sous-section.

.. et ceci un commentaire

Section 2
=========
La section 2 est ‘‘beaucoup plus‘‘ **intéressante** que la section 1.

Section 3
=========
La section 2 est un peu vantarde : la section 1 est *très bien*.

Une image au format "png"
~~~~~~~~~~~~~~~~~~~~~~~~~
.. figure:: helen.png
    :scale: 30%
```

L'utilitaire `rst2pdf`, appliqué à ce fichier, produit le fichier de même nom (☞ Fig. 10.3) mais avec l'extension `.pdf`.

FIGURE 10.3 – Exemple de sortie au format PDF

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

10.6.2 Le module `doctest`

Le principe du *literate programming* (ou programmation littéraire) de Donald Knuth consiste à mêler dans le source le code et la documentation du programme.

Ce principe a été repris en Python pour documenter les API *via* les chaînes de documentation (*docstrings*). Des programmes comme `Sphinx` peuvent alors les extraire des modules pour composer une documentation séparée.

Il est possible d'inclure dans les chaînes de documentation des exemples d'utilisation, écrits sous la forme de recopies de sessions interactives comportant ce que l'utilisateur saisit et le résultat produit par le logiciel.

Et pour aller plus loin, un module comme `doctest` est capable d'extraire ces exemples de sessions interactives et de les utiliser comme tests à exécuter pour vérifier le bon fonctionnement du logiciel.

Examinons deux exemples.

Pour chacun, nous donnerons d'une part le source muni de sa chaîne de documentation à partir de laquelle le module standard `doctest` permet d'extraire les parties sessions interactives puis de lancer ces sessions pour vérifier qu'elles fonctionnent, et d'autre part un résultat de l'exécution de ces tests.

Premier exemple : `documentation1.py`

```
# -*- coding: utf-8 -*-
"""Module d'essai de doctest."""

import doctest

def somme(a, b):
    """Renvoie a + b.

    >>> somme(2, 2)
    4
    >>> somme(2, 4)
    6
    """
    return a+b

if __name__ == '__main__':
    print("{:-^40}".format(" Mode silencieux "))
    doctest.testmod()
    print("Si tout va bien, on n'a rien vu !")

    print("\n{:-^40}".format(" Mode détaillé "))
    doctest.testmod(verbose=True)
```

L'exécution de ce fichier donne :

```
----------- Mode silencieux ------------
Si tout va bien, on n'a rien vu !

----------- Mode détaillé ------------
Trying:
    somme(2, 2)
Expecting:
```

Copyright © 2017 Dunod.

```
    4
ok
Trying:
    somme(2, 4)
Expecting:
    6
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.somme
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Deuxième exemple : `documentation2.py`

```
# -*- coding: UTF-8 -*-
"""Module d'essai de doctest."""

# fonctions
def accentEtrange(texte):
    """Ajoute un accent étrange à un texte.

    Les 'r' sont triplés, les 'e' suivis d'un 'u'

    Exemple :

    >>> texte = "Est-ce que tu as regardé la télé hier soir ? Il y avait un théma sur les
    ramasseurs d'escargots en Laponie, ils en bavent..."
    >>> print(accentEtrange(texte))
    Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr ? Il y avait un théma surRr leus
    rRramasseuurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...

    Cette technique permet d'internationaliser les applications
    pour les rendre compatibles avec certaines régions françaises.
    """
    texte = texte.replace('r', 'rRr')
    return texte.replace('e', 'eu')

def _test():
    import doctest
    doctest.testmod(verbose=True)

if __name__ == '__main__':
    _test()
```

Ce qui affiche :

```
Trying:
    texte = "Est-ce que tu as regardé la télé hier soir? Il y avait un théma sur les
ramasseurs d'escargots en Laponie, ils en bavent..."
Expecting nothing
ok
Trying:
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
    accentEtrange(texte)
Expecting:
    Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr? Il y avait un théma surRr leus
rRramasseuurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...
ok
2 items had no tests:
    __main__
    __main__._test
1 items passed all tests:
   2 tests in __main__.accentEtrange
2 tests in 3 items.
2 passed and 0 failed.
Test passed.
```

10.6.3 Le développement dirigé par la documentation

Comme on peut le voir, la documentation intégrée présente néanmoins un défaut : quand la documentation augmente, on ne voit plus le code !

La solution est de déporter cette documentation : la fonction `doctest.testfile()` permet d'indiquer le nom du fichier de documentation.

Qui plus est, on peut écrire ce fichier au format reST, ce qui permet de faire coup double. D'une part, on dispose des *tests intégrés* à la fonction (ou à la méthode) et, d'autre part, le même fichier fournit une *documentation* à jour.

Exemple : `test_documentation2.py`

Fichier de documentation [1] :

```
Le module ''accent''
====================

Test de la fonction ''accentEtrange''
-------------------------------------

Ce module fournit une fonction ''accentEtrange''.
On peut ainsi ajouter un accent à un texte :

        >>> from doctest2 import accentEtrange
        >>> texte = "Est-ce que tu as regardé la télé hier soir? Il y avait un théma sur
        les ramasseurs d'escargots en Laponie, ils en bavent..."
        >>> accentEtrange(texte)
        Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr? Il y avait un théma surRr
        leus rRramasseuurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...

Les ''r'' sont triplés et les ''e'' épaulés par des ''u''. Cette technique permet
de se passer de systèmes de traductions complexes pour faire fonctionner
les logiciels dans certaines régions.
```

1. Cf. [2], p. 217.

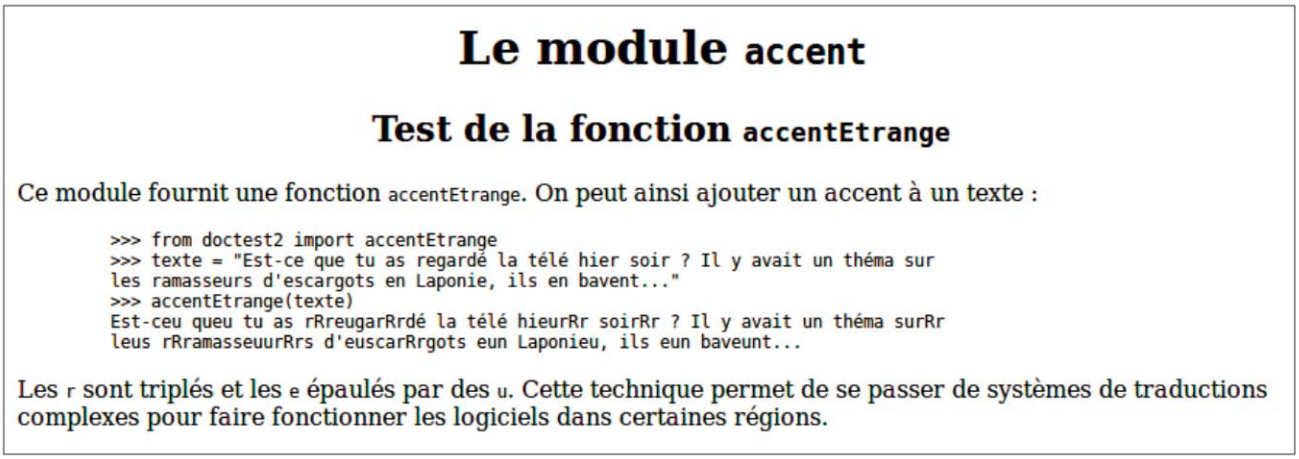
Copyright © 2017 Dunod.

Source du module :

```
import doctest
doctest.testfile("test_documentation2.txt", verbose=True)
```

Nous produisons la documentation HTML par la commande :

```
rst2html test_documentation2.txt test_documentation2.html
```



Le module accent

Test de la fonction accentEtrange

Ce module fournit une fonction accentEtrange. On peut ainsi ajouter un accent à un texte :

```
>>> from doctest2 import accentEtrange
>>> texte = "Est-ce que tu as regardé la télé hier soir ? Il y avait un théma sur
les ramasseurs d'escargots en Laponie, ils en bavent..."
>>> accentEtrange(texte)
Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr ? Il y avait un théma surRr
leus rRramasseuurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...
```

Les r sont triplés et les e épaulés par des u. Cette technique permet de se passer de systèmes de traductions complexes pour faire fonctionner les logiciels dans certaines régions.

FIGURE 10.4 – Documentation HTML du module `accent`

10.7 Exercices

1. ✠ Saisir un entier entre 1 et 3999. L'afficher sous forme de nombre romain en utilisant la fonction `zip()`.
   Voir les rappels sur la numérotation romaine p. 41.

2. ✠✠ Proposer une version récursive du problème des n dés présenté p. 52.

3. ✠ On donne une fonction `noncarres(n)` qui retourne la liste des nombres entiers de 1 à n qui ne sont pas des carrés de nombres entiers, en utilisant une boucle `while`. Celle-ci comporte plusieurs erreurs... Identifiez les problèmes et corrigez-les.

```
import math
def noncarres(n):
    lst = []
    i = 1
    while i <= n :
        reste = math.sqrt(i) - int(math.sqrt(i))
        if reste == 0:
            lst.append(i)
            i = i + 1
     print(lst)
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Solutions des exercices

Exercices du chapitre 2

```
# -*- coding: utf8 -*-
# Exercice 1

a >= 10 and a <=20

# ou

10 <= a <= 20
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Volume d'un cône droit."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from math import pi

# Programme principal ========================================================
rayon = float(input("Rayon du cône (m) : "))
hauteur = float(input("Hauteur du cône (m) : "))

volume = (pi*rayon*rayon*hauteur) / 3.0
print("Volume du cône =", volume, "m3")
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3

nbessais < 5 and (v <= 0 or v >= 100 or v % 3 != 0)
```

✦

```
# -*- coding: utf8 -*-
# Exercice 4

s = "Dark side of the moon"

s.title().center(60, '=')
```

✦

Exercices du chapitre 3

```
# -*- coding: utf8 -*-
# Exercice 1
```

Copyright © 2017 Dunod.

```
"""Calcul d'un prix TTC."""

# Programme principal ========================================================
prixHT = float(input("Prix HT (0 pour terminer)? "))
somTTC = 0
cptSup100 = 0
while prixHT > 0:
    prixTTC = prixHT * 1.2
    somTTC += prixTTC
    if prixTTC >= 100:
        cptSup100 += 1
    print("Prix TTC : {:.2f}\n".format(prixTTC))
    prixHT = float(input("Prix HT (0 pour terminer)? "))
print("Vous avez acheté pour {:.2f} € TTC d'articles.".formatg(somTTC))
print("Dans ceux-ci,", cptSup100, "valaient 100 € TTC ou plus.")
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Nombre de fois qu'un entier est divisible par 2."""

# Programme principal ========================================================
n = int(input("Entrez un entier strictement positif : "))
while n < 1:
    n = int(input("Entrez un entier STRICTEMENT POSITIF, s.v.p. : "))
save = n

cpt = 0
while n%2 == 0:
    n /= 2
    cpt += 1

print(save, "est", cpt, "fois divisible par 2.")
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3
"""Diviseurs propres d'un entier."""

# Programme principal ========================================================
n = int(input("Entrez un entier strictement positif : "))
while n < 1:
    n = int(input("Entrez un entier STRICTEMENT POSITIF, s.v.p. : "))

i = 2       # plus petit diviseur possible de n
cpt = 0     # initialise le compteur de divisions
p = n/2     # calculé une fois dans la boucle

print("Diviseurs propres sans répétition de ", n, ":", end=' ')
while i <= p:
    if n%i == 0:
        cpt += 1
        print(i, end=' ')
    i += 1
```

Copyright © 2017 Dunod.

```
if not cpt:
    print("aucun ! C'est un nombre premier.")
else:
    print("(soit", cpt, "diviseurs propres)")
```

✦

```
# -*- coding: utf8 -*-
# Exercice 4
"""Nombres romains (version 1)."""

# Programme principal ========================================================
n = int(input('Entrez un entier [1 .. 4000[ : '))
while not(n >= 1 and n < 4000):
    n = int(input('Entrez un entier [1 .. 4000[, s.v.p. : '))

s = ""  # Chaîne résultante

while n >= 1000:
    s += "M"
    n -= 1000

if n >= 900:
    s += "CM"
    n -= 900

if n >= 500:
    s += "D"
    n -= 500

if n >= 400:
    s += "CD"
    n -= 400

while n >= 100:
    s += "C"
    n -= 100

if n >= 90:
    s += "XC"
    n -= 90

if n >= 50:
    s += "L"
    n -= 50

if n >= 40:
    s += "XL"
    n -= 40

while n >= 10:
    s += "X"
    n -= 10

if n >= 9:
    s += "IX"
    n -= 9
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
if n >= 5:
    s += "V"
    n -= 5

if n >= 4:
    s += "IV"
    n -= 4

while n >= 1:
    s += "I"
    n -= 1

print("En romain :", s)
```

✦

Exercices du chapitre 4

```
# -*- coding: utf8 -*-
# Exercice 1
"""Mélange de Monge"""

# Programme principal ========================================================
n, monge = 5, [1]

print('Paquet initial   :', list(range(1, 2*n+1)))

for i in range(2, 2*n+1):
    if i%2 == 0:  #indice pair
        monge.insert(0, i)
    else:
        monge.append(i)

print('Mélange de Monge :', monge)
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Jeu de dés (1)."""

# Programme principal ========================================================
n = int(input("Entrez un entier [2 .. 12] : "))
while not(n >= 2 and n <= 12):
    n = int(input("Entrez un entier [2 .. 12], s.v.p. : "))

s = 0
for i in range(1, 7):
    for j in range(1, 7):
        if i+j == n:
            s += 1

print("Il y a {:d} façon(s) de faire {:d} avec deux dés.".format(s, n))
```

✦

Copyright © 2017 Dunod.

```
# -*- coding: utf8 -*-
# Exercice 3
"""Jeu de dés (2)."""

# Programme principal ============================================================
n = int(input("Entrez un entier [3 .. 18] : "))
while not(n >= 3 and n <= 18):
    n = int(input("Entrez un entier [3 .. 18], s.v.p. : "))

s = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            if i+j+k == n:
                s += 1

print("Il y a {:d} façon(s) de faire {:d} avec trois dés.".format(s, n))
```

```
# -*- coding: utf8 -*-
# Exercice 4
"""Jeu de dés (3)."""

# Globale ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MAX = 8

# Programme principal ============================================================
nbd = int(input("Nombre de dés [2 .. {:d}] : ".format(MAX)))
while not(nbd >= 2and nbd <= MAX):
    nbd = int(input("Nombre de dés [2 .. {:d}], s.v.p. : ".format(MAX)))

s = int(input("Entrez un entier [{:d} .. {:d}] : ".format(nbd, 6*nbd)))
while not(s >= nbd and s <= 6*nbd):
    s = int(input("Entrez un entier [{:d} .. {:d}], s.v.p. : ".format(nbd, 6*nbd)))

if s == nbd or s == 6*nbd:
    cpt = 1 # 1 seule solution
else:
    init = [1]*nbd      # initialise une liste de <nbd> dés
    cpt, j = 0, 0
    while j < nbd:
        som = sum([init[k] for k in range(nbd)])

        if som == s:
            cpt += 1    # compteur de bonnes solutions
        if som == 6*nbd:
            break

        j = 0
        if init[j] < 6:
            init[j] += 1
        else:
            while init[j] == 6:
                init[j] = 1
                j += 1
            init[j] += 1
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
print("Il y a {:d} façons de faire {:d} avec {:d} dés.".format(cpt, s, nbd))
```

✦

Exercices du chapitre 5

```
# -*- coding: utf8 -*-
# Exercice 1
"""Enregistrement masses volumiques des éléments."""

lst = [('Arsenic', 17.8464, 3.12), ('Aluminium', 16.767, 6.21), ('Or', 239320, 12400)]

f = open("elements.txt", "w", encoding="utf-8")
for nom, masse, volume in lst:
    mvol = masse / volume
    s = "{} = {:.2f} g/cm3\n".format(nom, mvol)
    f.write(s)
f.close()
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Affichage de la masse d'un volume suivant les éléments."""

volume = float(input("Volume de matière (cm3) : "))

f = open("elements.txt", "r", encoding="utf-8")
for ligne in f:
    items = ligne.split("=")
    nom = items[0].strip()
    mvol = float(items[1].split()[0])
    masse = volume * mvol
    print("{} : {:.2f} g".format(nom, masse))
f.close()
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3
"""Journalisation de messages."""

from time import asctime

def note_journal(nomfichier, message):
    "Stockage d'un message horodaté."
    f = open(nomfichier, "a", encoding="utf-8")
    f.write(asctime())
    f.write(" " + message + "\n")
    f.close()

# Tests
import time
for i in range(10):
    note_journal("jourheure.txt", "Un message {}.".format(i+1))
    time.sleep(1.5)
```

Copyright © 2017 Dunod.

✦

Exercices du chapitre 6

```
# -*- coding: utf8 -*-
# Exercice 1
"""Approximation de 'e'."""

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def fact(n):
    r = 1
    for i in range(1, n+1):
        r *= i
    return r

# Programme principal ========================================================
n = int(input("n ? "))
exp = 0.0
for i in range(n):
    exp = exp + 1.0/fact(i)

print("Approximation de 'e' : {:.3f}".format(exp))
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Gardien de phare."""

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def hauteurParcourue(n, nb, h):
    print("Pour {:d} tours de maintenance et {:d} marches de {:d} cm,"
          " il parcourt un dénivelé de {:.2f} m par semaine !"
          .format(n, nb, h, n * nb * h * 2 * 7 / 100.0))

# Programme principal ========================================================
nb_tours = int(input("Combien de tours de maintenance ? "))
nb_marches = int(input("Combien de marches ? "))
hauteur_marche = int(input("Hauteur d'une marche (cm) ? "))

hauteurParcourue(nb_tours, nb_marches, hauteur_marche)
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3
"""Min, max et moyenne d'une liste d'entiers."""

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def minMaxMoy(liste):
    """Renvoie le min, le max et la moyenne de la liste."""
    min, max, som = liste[0], liste[0], float(liste[0])
    for i in liste[1:]:
        if i < min:
            min = i
        if i > max:
            max = i
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
        som += i
    return (min, max, som/len(liste))

# Programme principal =========================================================
lp = [10, 18, 14, 20, 12, 16]

print("liste =", lp)
l = minMaxMoy(lp)
print("min : {0[0]}, max : {0[1]}, moy : {0[2]}".format(l))
```

✦

Exercices du chapitre 7

```
# -*- coding: utf8 -*-
# Exercice 1 (1)
"""Module de tracé de polygones réguliers avec Turtle."""

# import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from turtle import forward, left, width, done


# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def polygone_regulier(ncotes, longueur):
    angle = 360/ncotes
    for i in range(ncotes):
        forward(longueur)
        left(angle)


# Auto-test ===================================================================
if __name__=='__main__':
    width(3)

    polygone_regulier(3, 300)
    polygone_regulier(4, 80)
    polygone_regulier(8, 100)
    polygone_regulier(100, 5)

    done()
```

✦

```
# -*- coding: utf8 -*-
# Exercice 1 (2)
"""Tracé de polygones avec Turtle."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from turtle import left, width, done
from turtle_m import polygone_regulier

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def polygones(npoly, ncotes, longueur):
    rot = 360 / npoly
    for i in range(npoly):
        polygone_regulier(ncotes, longueur)
        left(rot)
```

Copyright © 2017 Dunod.

```
# programme principal ========================================================
width(3)

polygones(10, 3, 150)

done()
```

✦

```
# -*- coding: utf8 -*-
# Exercice 2
"""Liste d'entiers différents."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from random import seed, randint

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires entre <a> et <b>."""
    return [randint(a, b) for i in range(n)]

# Programme principal ========================================================
n = int(input("Entrez un entier [1 .. 100] : "))
while not(n >= 1 and n <= 100):
    n = int(input("Entrez un entier [1 .. 100], s.v.p. : "))

# Construction de la liste
seed()  # initialise le générateur de nombres aléatoires
t = listAleaInt(n, 0, 500)

# Sont-ils différents ?
tousDiff = True
i = 0
while tousDiff and i < (n-1):
    j = i + 1
    while tousDiff and j < n:
        if t[i] == t[j]:
            tousDiff = False
        else:
            j += 1
    i += 1

if tousDiff:
    print("\nTous les éléments sont distincts.")
else:
    print("\nAu moins une valeur est répétée.")
print(t)
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3
"""Liste d'entiers différents (seconde version)."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from random import seed, randint
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires entre <a> et <b>."""
    return [randint(a, b) for i in range(n)]

# Programme principal ========================================================
n = int(input("Entrez un entier [1 .. 100] : "))
while not(n >= 1 and n <= 100):
    n = int(input("Entrez un entier [1 .. 100], s.v.p. : "))

seed()  # initialise le générateur de nombres aléatoires
avant = listAleaInt(n, 0, 500)
apres = list(set(avant))

if len(avant) == len(apres):
    print("\nTous les éléments sont distincts.")
else:
    print("\nAu moins une valeur est répétée.")
print(avant)
```

```
# -*- coding: utf8 -*-
# Exercice 4 (1)
"""Module pour les nombres parfaits et chanceux."""

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def somDiv(n):
    """Retourne la somme des diviseurs propres de <n>."""
    som_div = 1
    for div in range(2, (n//2)+1):
        if n % div == 0:
            som_div += div
    return som_div


def estParfait(n):
    """Retourne True si <n> est parfait, False sinon."""
    return somDiv(n) == n


def estPremier(n):
    """Retourne True si <n> est premier, False sinon."""
    return somDiv(n) == 1


def estChanceux(n):
    """Retourne True si <n> est chanceux, False sinon."""
    est_chanceux = True
    for i in range(0, n-1):
        est_chanceux = est_chanceux and estPremier(n + i + i*i)
    return est_chanceux

# Auto-test ==================================================================
if __name__=='__main__':
    from math import isclose
```

Copyright © 2017 Dunod.

```
    print(isclose(somDiv(12), 16))
    print(isclose(estParfait(6), True))
    print(isclose(estPremier(31), True))
    print(isclose(estChanceux(11), True))
```

✦

```
# -*- coding: utf8 -*-
# Exercice 4 (2)
"""Nombres chanceux et parfaits."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from parfait_chanceux_m import estParfait, estChanceux

# programme principal ========================================================
parfait, chanceux = [], []

for n in range(2, 1001):
    if estParfait(n):
        parfait.append(n)
    if estChanceux(n):
        chanceux.append(n)

print("\nIl y a {} nombres parfaits dans [2, 1000] : {}".format(len(parfait), parfait))
print("\nIl y a {} nombres chanceux dans [2, 1000] : {}".format(len(chanceux), chanceux))
```

✦

Exercice du chapitre 8

```
# -*- coding: utf8 -*-
# Exercice 1
import random

class Domino:
    def __init__(self, facea, faceb):
        self.fa = facea
        self.fb = faceb

    def __str__(self):
        return "[{}:{}]".format(self.fa, self.fb)

    def appariement(self, autre):
        "Retourne la marque du premier côté apparié trouvé, sinon None."
        for n in (self.fa, self.fb):
            if n in (autre.fa, autre.fb):
                return n
        return None

pioche = []
for i in range(0, 7):
    for j in range(0, 7):
        pioche.append(Domino(i, j))
random.shuffle(pioche)
joueur1 = pioche[0:7]
del pioche[0:7]
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
joueur2 = pioche[0:7]
del pioche[0:7]

print("Dominos joueur 1 : ", end='')
for d in joueur1:
    print(d, end='  ')
print()
print("Dominos joueur 2 : ", end='')
for d in joueur2:
    print(d, end='  ')

d1 = joueur1[0]
for d2 in joueur2:
    if d1.appariement(d2) is not None:
        print(d1, "<==>", d2)
```

✦

Exercices du chapitre 9

```
# -*- coding: utf8 -*-
# Exercice 1
import tkinter as tk

fen = tk.Tk()
lignezero = tk.Frame(fen)
lignezero.pack(side=tk.TOP, fill='x', expand=1)
etiq = tk.Label(lignezero, text="Valeur:")
etiq.pack(side=tk.LEFT)
chp = tk.Entry(lignezero)
chp.pack(side=tk.LEFT)
ccocher = tk.Checkbutton(fen, text="Toujours utiliser cette valeur")
ccocher.pack(side=tk.TOP)
lignedeux = tk.Frame(fen)
lignedeux.pack(side=tk.TOP)
btnok = tk.Button(lignedeux, text="Ok")
btnok.pack(side=tk.LEFT)
btnann = tk.Button(lignedeux, text="Annuler")
btnann.pack(side=tk.LEFT)
fen.mainloop()
```

✦

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Exercice 2

import tkinter as tk
root = tk.Tk()
tk.Label(root, text="Nom         ").grid(row=0, sticky=tk.W)
tk.Label(root, text="Mot de passe").grid(row=1, sticky=tk.W)
tk.Entry(root).grid(row=0, column=1, sticky=tk.E)
tk.Entry(root).grid(row=1, column=1, sticky=tk.E)
tk.Button(root, text="Login", command=root.destroy).grid(row=2, column=1, sticky=tk.E)
root.mainloop()
```

✦

Copyright © 2017 Dunod.

Exercices du chapitre 10

```
# -*- coding: utf8 -*-
# Exercice 1
"""Nombres romains (version 2)."""

# globales ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CODE = zip(
    [1000,900 ,500,400 ,100,90  ,50 ,40  ,10 ,9   ,5  ,4   ,1],
    ["M" ,"CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"]
)

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def decToRoman(num):
    res = []
    for d, r in CODE:
        while num >= d:
            res.append(r)
            num -= d
    return ''.join(res)

# Programme principal ========================================================
for i in range(1, 4000):
    print(i, decToRoman(i))
```

```
# -*- coding: utf8 -*-
# Exercice 2
"""Jeu de dés (récursif)."""

# Globale ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MAX = 8

# Définition de fonction ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def calcul(d, n):
    """Calcul récursif du nombre de façons de faire <n> avec <d> dés."""
    resultat, debut = 0, 1
    if (d == 1) or (n == d) or (n == 6*d):  # conditions terminales
        return 1
    else:                   # sinon appels récursifs
        if n > 6*(d-1):     # optimisation importante
            debut = n - 6*(d-1)

        for i in range(debut, 7):
            if n == i:
                break
            resultat += calcul(d-1, n-i)
    return resultat

# Programme principal ========================================================
d = int(input("Nombre de dés [2 .. {:d}] : ".format(MAX)))
while not(d >= 2and d <= MAX):
    d = int(input("Nombre de dés [2 .. {:d}], s.v.p. : ".format(MAX)))

n = int(input("Entrez un entier [{:d} .. {:d}] : ".format(d, 6*d)))
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
while not(n >= d and n <= 6*d):
    n = int(input("Entrez un entier [{:d} .. {:d}], s.v.p. : ".format(d, 6*d)))

print("Il y a {:d} façon(s) de faire {:d} avec {:d} dés.".format(calcul(d, n), n, d))
```

✦

```
# -*- coding: utf8 -*-
# Exercice 3
import math
def noncarres(n):
    lst = []
    i = 1
    while i <= n :
        reste = math.sqrt(i) - int(math.sqrt(i))
        if reste != 0:          # condition était inversée
            lst.append(i)
        i = i + 1               # incrémentation était hors de la boucle
     return lst                 # print était utilisé

"""
Trois bugs à corriger :
  * la condition du if est inversée → if reste!= 0 ;
  * l'incrémentation du i doit être faite systématiquement, dans le corps du while, sinon on a une
    boucle ∞ → désindenter le i=i+1 pour qu'il soit au même niveau que le if ;
  * la fonction doit retourner la liste, pas l'afficher → dernière ligne return lst.
"""
```

Copyright © 2017 Dunod.

Annexe A

Interlude

Le Zen de Python [1]

Préfère
la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe,
le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,
Ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait y avoir qu'une et une seule façon de procéder.
Même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de noms, sacrée bonne idée ! Faisons plus de trucs comme ça !

1. `import this` de Tim Peters (PEP n° 20), traduction Cécile Trevian et Bob Cordeau.

Copyright © 2017 Dunod.

Copyright © 2017 Dunod.

Annexe B

La distribution Pyzo et Jupyter Notebook

Parmi la multitude d'environnements de développement permettant de travailler avec Python, le choix de `Pyzo` avec `miniconda` est particulièrement pertinent pour l'initiation. Cette distribution comprend un environnement de développement intégré simple et complet, les principales bibliothèques scientifiques (`Numpy`, `Scipy`, `Matplotlib`...) et graphiques (`tkinter`, `pyQt`) ainsi que `conda`, un gestionnaire de paquets performant.

Introduction

`Pyzo` existe pour les trois principaux systèmes d'exploitation : Windows, Mac OS X et GNU/Linux.

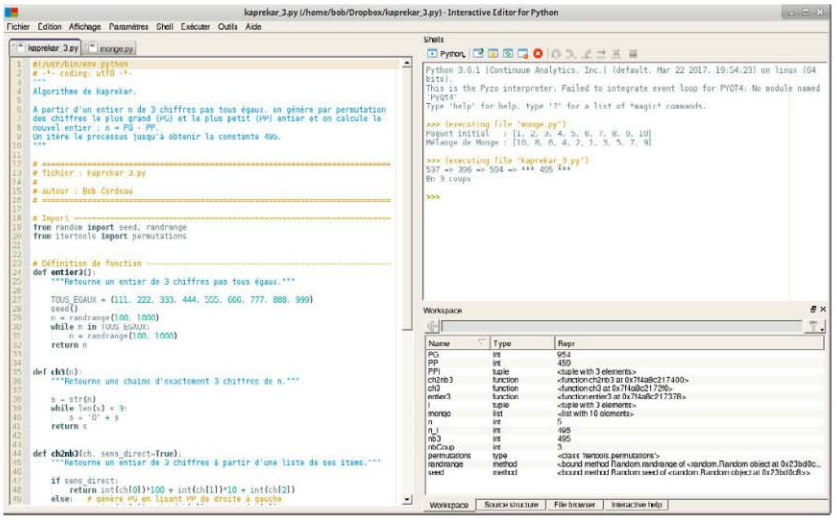


Figure B.1 – Exemple de configuration de l'EDI `Pyzo`

L'EDI `Pyzo` permet d'afficher des scripts en profitant de la coloration syntaxique, d'utiliser indépendamment un interpréteur (sorties des scripts, débogage, documentation, test de fragments de code...) et d'agencer interactivement divers outils (☞ Fig. B.1).

De plus, le gestionnaire de paquets `conda` inclus dans la distribution `miniconda` est utilisable directement **dans** l'interpréteur de l'EDI.

Copyright © 2017 Dunod.

`Jupyter Notebook` [1] est un environnement de calcul interactif accessible depuis un navigateur web, dans lequel on peut aisément combiner du code, du texte simple ou enrichi de tableaux ou de notations mathématiques, des tracés de courbes, des vidéos, etc. Il permet de travailler en mode collaboratif à plusieurs utilisateurs sur un même document partagé. Des plateformes d'hébergement comme `nbviewer` (`https://nbviewer.jupyter.org/`) ou `SageMathCloud` (`https://cloud.sagemath.com/`) autorisent de tels partages.

Cette pratique permet d'inclure naturellement tous les documents qui accompagnent un code et qui sont habituellement séparés (et souvent dans un format différent !) du code.

Ce format standard [2] permet de réaliser des documents complets, de les sauvegarder ou les charger facilement donc de les partager. Grâces aux *notebooks*, on peut « jouer » avec les codes en les modifiant autant que l'on veut : c'est la base de la pédagogie.

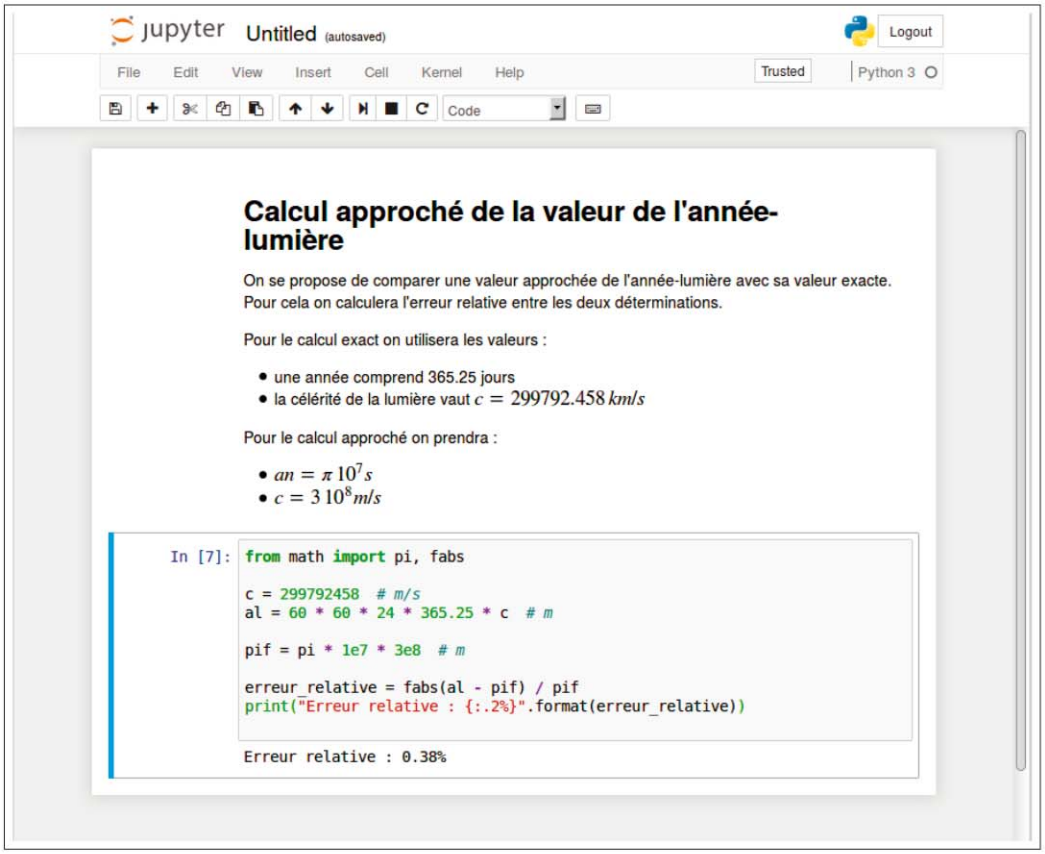


FIGURE B.2 – Exemple d'utilisation de `Jupyter Notebook`

Installation

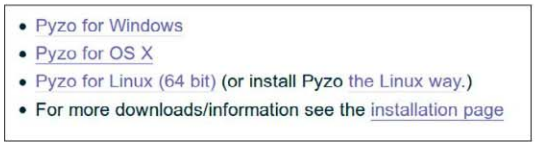
L'installation de cet environnement comporte trois étapes [3].

1. Cf. une démonstration simple à l'adresses `https://try.jupyter.org`
2. Les *notebooks* sont sauvegardés au format `json`.
3. L'installation est détaillée à l'adresse `http://www.pyzo.org/start.html#quickstart`

Copyright © 2017 Dunod.

Étape 1 : installation de l'EDI Pyzo

Les applications sont disponibles sur le site http://www.pyzo.org/start.html#quickstart. Choisissez votre système d'exploitation :



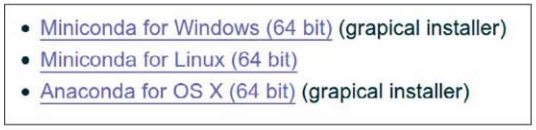
Remarque 1 Pour obtenir l'installeur pour d'autres architectures, suivez le lien `installation page` puis, dans le haut de la page, le lien `all available binaries`.

Remarque 2 Vous pouvez installer `Pyzo` sous votre propre compte, il n'est pas nécessaire d'être l'administrateur de votre machine.

Étape 2 : installation de l'environnement Python 3

La distribution `miniconda3` offre la dernière version recommandée de Python 3, une sélection de paquets supplémentaires et `conda`, l'outil de gestion de la distribution utilisable dans `Pyzo`.

Toujours sur le même site et sous votre compte (prévoyez un espace de stockage d'environ 2 Go si vous installez les paquets indiqués ci-après), sélectionnez votre système et installez-le.



Encore une fois, d'autres architectures sont disponibles en suivant le lien `miniconda`.

Étape 3 : installation des paquets scientifiques

Pour tirer le meilleur parti de ce livre ainsi que pour étudier les exercices proposés sur le site https://www.dunod.com/sciences-techniques/python-3, il est nécessaire d'ajouter quelques paquets scientifiques.

Un grand intérêt de `Pyzo` est de pouvoir utiliser `conda` directement dans son EDI. Pour cela on lance `Pyzo` et, dans sa fenêtre `Shells`, on utilise le gestionnaire de paquet `conda` (il est aussi possible d'utiliser `pip` pour les paquets qui ne seraient pas disponibles *via* `conda`).

Les principales commandes de gestion de conda

Pour apprécier la puissance de `conda`, intéressons-nous aux commandes qu'il fournit. Nous serons alors à même de gérer complètement notre distribution Python.

`conda` est un gestionnaire de paquets *open source* écrit en Python. Il fonctionne sur Linux, OS X et Windows. Plus de 720 paquets scientifiques peuvent être installés individuellement par la commande `conda install`.

Nous allons donner les commandes de `conda` les plus utiles pour installer et mettre à jour son installation. Une documentation complète est disponible en ligne [1].

1. https://conda.io/docs/using/index.html

© Dunod. Toute reproduction non autorisée est un délit.
Copyright © 2017 Dunod.

Pour chaque commande, une aide est fournie. Par exemple : `>>> conda update --help`.

Enfin, avant d'installer ou de mettre à jour des paquets, il est recommandé de vérifier la version des logiciels :

```
>>> conda info
```

Principales commandes `conda`	Signification
`conda list`	liste les paquets de l'installation
`conda search pkg`	vérifie si le paquet `pkg` est disponible
`conda search -f python`	vérifie les versions de Python disponibles
`conda install pkg`	installe le paquet `pkg`
`conda update pkg`	met à jour le paquet `pkg`
`conda update –all`	met à jour tous les paquets installés
`conda remove pkg`	enlève le paquet `pkg` de l'installation

Pour finir l'installation

Compte tenu des informations données ci-dessus, l'installation des paquets scientifiques requiert les commandes :

```
>>> conda update conda
>>> conda install numpy scipy pyqt matplotlib pandas sympy ipywidgets notebook
```

Remarque

✓ Si la version de Python évolue, il est plus simple de réinstaller `miniconda` puis les paquets, que de mettre en place un nouvel environnement complet avec `conda create`.

Utilisation de `Pyzo`

La configuration de base de `Pyzo` est limitée au choix de la police de caractères de l'éditeur (menu `Affichage/Police de caractères`) et au choix de la langue de l'EDI (menu `Paramètres/Sélectionner la langue`).

La fenêtre de `Shell`

Le shell (ou interpréteur) par défaut de `Pyzo` est un shell « amélioré » au sens où il comprend des commandes dites *magic*. On peut afficher leur liste en tapant `?`.

Mais si on préfère utiliser le shell `IPython`, ce qui se justifie particulièrement dans le domaine scientifique, on peut facilement le configurer. Le menu `Shell/Configuration des shells` permet soit d'ajouter un nouveau shell, soit de modifier le shell par défaut.

Par exemple, pour configurer IPython en tant que shell supplémentaire, il suffit de cliquer sur `Ajouter une configuration`, de donner un nom au nouvel onglet créé et de cocher `Utilise IPython s'il est disponible`. Enfin dans la fenêtre Shell, on clique sur `Python/Nouveau shell...` et on choisit son interpréteur (ou raccourcis `Ctrl+1` et `Ctrl+2`).

Copyright © 2017 Dunod.

L'éditeur

L'éditeur offre la coloration syntaxique qui rend le code plus lisible. Lorsque l'on tape le nom d'une fonction suivi d'une parenthèse ouvrante, une bulle d'aide syntaxique s'ouvre automatiquement.

Pour exécuter un script, plusieurs solutions existent. En fonction de ses besoins, on peut soit utiliser le menu `Exécuter/Démarrer le script` (raccourci : `Ctrl+Shift+E`) pour un module [1], soit le menu `Exécuter/Démarrer le script principal` (raccourci : `Ctrl+Shift+M`) pour un programme principal. Si votre projet comporte plusieurs fichiers, le script principal est sélectionné par le menu contextuel accessible par clic-droit sur l'onglet du fichier choisi. Le résultat des exécutions avec un interpréteur Python réinitialisé s'affiche dans la fenêtre Shell.

Une caractéristique de `Pyzo` est la *cellule*. On définit une cellule en délimitant une partie du code de l'éditeur par `##` (suivi éventuellement d'un nom de cellule). Lorsque le curseur se trouve dans une cellule, on peut l'exécuter en cliquant sur le menu `Exécuter/Exécuter la cellule` ou en tapant le raccourci `Ctrl+Entrée`, son exécution est réalisée sans réinitialiser l'interpréteur Python (les noms déjà définis, les modules déjà chargés restent accessibles). Cette possibilité est particulièrement intéressante dans un cadre d'enseignement, voire de débogage.

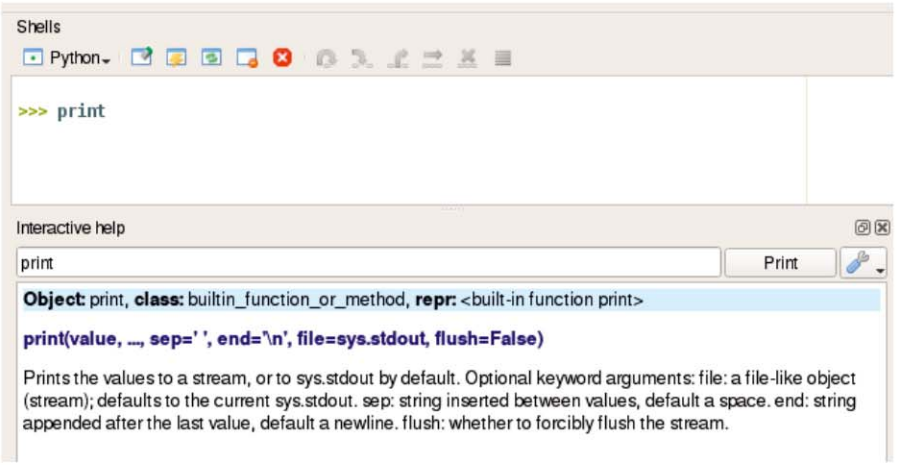


FIGURE B.3 – Outil `Interactive help` (exemple de la fonction `print()`)

Les outils

Parmi les outils disponibles de l'IDE (menu `Outils`), quatre sont recommandés :

- **Workspace** : cette fenêtre affiche tous les noms définis par les exécutions depuis le démarrage de l'interpréteur. Son utilisation est indispensable en cas de débogage [cf. p. 194] ;
- **Source structure** : cette fenêtre affiche le « squelette » du script en cours d'édition (☞ Fig. B.4), ainsi que les cellules définies par `## nom` ;
- **File browser** : un navigateur de fichiers ;
- **Interactive help** : fenêtre de documentation en interaction avec la fenêtre Shell et la fenêtre d'édition (☞ Fig. B.3).

1. Les habitués d'IDLE seront tentés d'utiliser le raccourci caché `F5`. Mais attention, `F5` réexécute le module édité dans le contexte de l'interpréteur en cours. Si on a modifié un autre module qui avait déjà été chargé précédemment, il n'est pas rechargé.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Signalons également que le menu **Aide** de Pyzo fournit un guide en ligne [1], ainsi qu'un guide de démarrage rapide.

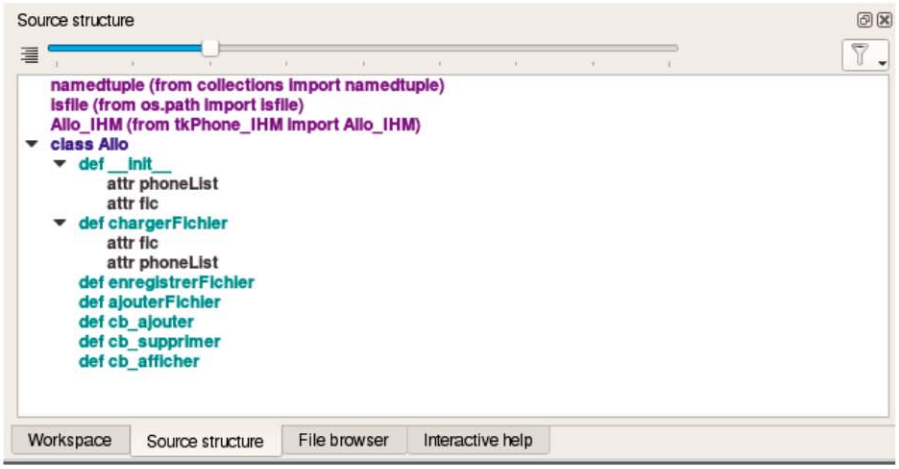


FIGURE B.4 – Outil **Source structure** (exemple du source **tkPhone.py** [cf. p. 115])

Utilisation de Jupyter

Lancement de Jupyter

Comme nous l'avons vu, il existe une forte cohérence entre les applications utilisées dans ce livre : la commande **notebook** du shell de **Pyzo** lance en effet l'application **Jupyter Notebook**.

Attention

☢ **Jupyter** fonctionne en tant que processus serveur web sur votre ordinateur. Toute personne qui peut s'y connecter a la possibilité d'exécuter n'importe quel programme et d'accéder à toutes vos données. Il est important de sécuriser cet accès.

Par défaut, **Jupyter** démarre en fournissant un lien comportant un code numérique propre à chaque session. Montrons, ci-dessous, comment remplacer ce code abscons par un mot de passe personnel qui sera fixé une fois pour toutes, avec la commande **notebook passwd** :

```
>>> notebook password
/home/bob/miniconda3/lib/python3.6/getpass.py:62: GetPassWarning:
                                                  Can not control echo on the terminal.
  passwd = fallback_getpass(prompt, stream)
Warning: Password input may be echoed.
Enter password: toto

Warning: Password input may be echoed.
Verify password: toto

[NotebookPasswordApp] Wrote hashed password to /home/bob/.jupyter/jupyter_notebook_config.json
```

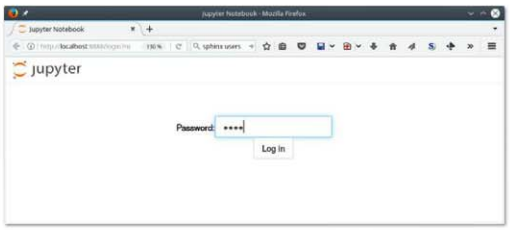
On peut ensuite démarrer simplement **Jupyter** avec la commande **notebook** :

1. En anglais : http://www.pyzo.org/guide.html

Copyright © 2017 Dunod.

```
>>> notebook
[I 16:45:31.800 NotebookApp] Writing notebook server cookie secret to
                             /run/user/10501/jupyter/notebook_cookie_secret
[I 16:45:31.851 NotebookApp] Serving notebooks from local directory: /home/bob
[I 16:45:31.862 NotebookApp] 0 active kernels
[I 16:45:31.872 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 16:45:31.882 NotebookApp] Use Control-C to stop this server and shut down all kernels
                                                    (twice to skip confirmation).
[I 16:45:32.022 NotebookApp] 302 GET /tree (127.0.0.1) 1.30ms

************** ci-dessous la trace de connexion via l'interface web ***********

[I 16:48:11.897 NotebookApp] 302 POST /login?next=%2Ftree (127.0.0.1) 1.03ms
```

Une fois le mot de passe en place (un peu moins attendu que `toto` bien sûr !), il suffit d'ouvrir dans votre navigateur l'url par défaut `http://localhost:8888/` pour arriver sur l'écran de connexion et saisir votre mot de passe.



Votre navigateur web s'affiche et présente la liste des fichiers du répertoire de lancement. Les `notebooks` sont repérés par l'icône d'un petit carnet (l'icône est verte si le notebook est lancé). Ils s'ouvrent grâce à un seul clic.

Utilisation du notebook

En haut et à droite de l'application, le bouton `New` permet le lancer un nouveau notebook avec l'interpréteur Python 3.

Un notebook est constitué d'une succession de *cellules* pouvant contenir deux types d'informations, soit du code Python, soit du texte enrichi avec la syntaxe `markdown`. Les cellules peuvent être dans un mode commande (liseré bleu) ou dans un mode édition (liseré vert).

Le passage du mode commande au mode édition se fait en cliquant sur une cellule Python, ou en double-cliquant sur une cellule markdown.

Le passage du mode édition au mode commande se fait par l'un des raccourcis suivants :

- `Ctrl-Enter` : exécute la cellule ;
- `Shift-Enter` : exécute la cellule et sélectionne la cellule suivante. L'appui répété de cette touche permet ainsi d'exécuter pas à pas toutes les cellules du notebook ;
- `Alt-Enter` : exécute la cellule et insère une nouvelle cellule juste en dessous.

L'exécution d'une cellule Python interprète le code. L'exécution d'une cellule markdown affiche le rendu du texte balisé.

Les cellules markdown

Elles sont utilisées pour produire du texte *enrichi* grâce au langage *markdown*, qui permet de produire simplement des mises en page comprenant textes avec styles, tableaux, hiérarchie de titres,

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

images, vidéos, formules mathématiques... Nous ne donnerons pas ici le détail de la syntaxe, qui est disponible en ligne sur le site `http://daringfireball.net/projects/markdown/`.

Les cellules Python

Elles permettent d'écrire du code Python et sont repérables par le prompt `In[n]` où le numéro entre crochets est un compteur incrémenté à chaque sollicitation de l'interpréteur. Les cellules reçoivent du code Python 3 (comme c'est rappelé en haut et à droite de la fenêtre). Les scripts peuvent utiliser toute la puissance de Python, même les entrées/sorties (`input()`/`print()`) sont gérées par l'interface pour utiliser les interactions du navigateur web, ainsi que l'affichage des graphes par `matplotlib`.

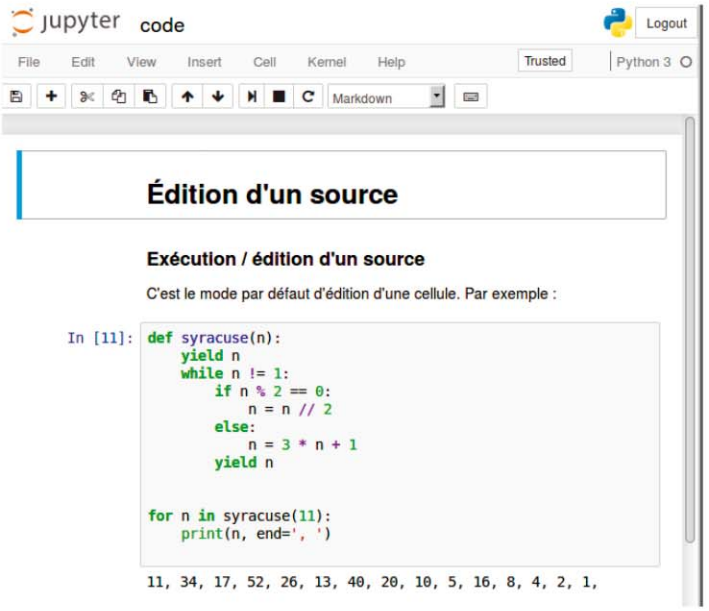


FIGURE B.5 – Utilisation de `Jupyter Notebook`

Avec l'installation du paquet `ipywidgets`, il est possible de mettre en place des scripts et des graphiques qui comportent une zone de contrôle utilisable pour faire varier des paramètres et voir immédiatement le résultat s'afficher [cf. p. 84].

L'aide en ligne

Chaque notebook affiche une barre de menu qui comporte un bouton `Help`. On y trouve principalement :

- `User Interface Tour` : survol graphique des éléments de l'interface ;
- `Keyboard Shortcuts` : tableau des raccourcis des modes commande et édition ;
- `Notebook Help` : didacticiels et exemples ;
- `Markdown` : le langage markdown ;
- les liens des documentations `Python`, `IPython`, `NumPy`, `SciPy`, `Matplotlib`, `SimPy` et `pandas`.

Copyright © 2017 Dunod.

Annexe C

Jeux de caractères et encodage

Position du problème

Nous avons vu que l'ordinateur code toutes les informations qu'il manipule en *binaire*. Pour coder les nombres entiers, un changement de base suffit ; pour les flottants, on utilise une norme (IEEE 754), mais la situation est plus complexe pour représenter les caractères.

Tous les caractères que l'on peut écrire à l'aide d'un ordinateur sont représentés en mémoire par des nombres. On parle de *codage*. Le « a » minuscule par exemple peut être représenté, ou codé, par le nombre 97. Pour pouvoir afficher ou imprimer un caractère lisible, ses différents dessins, appelés *glyphes*, sont stockés dans des catalogues appelés *polices de caractères*. Les logiciels informatiques parcourent ces catalogues pour rechercher le glyphe qui correspond à un nombre. Suivant la police de caractères, on peut ainsi afficher différents aspects du même « a » (97).

Les 128 premiers caractères comprennent des caractères de contrôle, les caractères de l'alphabet latin (non altérés [1]), les majuscules et les minuscules, les chiffres arabes, et quelques signes de ponctuation, c'est la fameuse table ASCII [2] (☞ Fig. C.1). Il y a longtemps, chaque pays avait complété ce jeu initial suivant les besoins de sa propre langue, créant ainsi son propre système de codage.

Ces définitions locales ont un fâcheux inconvénient : le caractère « à » français peut alors être représenté par le même nombre que le caractère « å » scandinave dans les deux codages, ce qui rend impossible l'écriture d'un texte bilingue avec ces deux caractères !

Le codage Unicode

Pour écrire un document en plusieurs langues, le standard nommé Unicode a donc été développé et est maintenu par un consortium international [3]. Il permet d'unifier une grande table de correspondance, sans chevauchement entre les caractères. Les catalogues de police se chargent ensuite de fournir des glyphes correspondants.

L'encodage UTF-8

Comme il s'agit de différencier plusieurs centaines de milliers de caractères (on compte plus de 6 000 langues dans le monde), il n'est évidemment pas possible de les encoder sur un seul octet.

En fait, la norme Unicode spécifie seulement le code numérique de l'identifiant associé à chaque caractère (☞ Fig. C.2). Elle ne force pas un format particulier de stockage, laissant libre le nombre d'octets ou de bits à réserver pour l'encodage. Elle définit toutefois des encodages de stockage normalisés.

Comme la plupart des textes produits en Occident utilisent essentiellement la table ASCII, qui correspond justement à la partie basse de la table Unicode [4], l'encodage le plus économique est l'UTF8 :

1. C'est-à-dire sans signe diacritique, par exemple les accents, le tréma, la cédille…
2. American Standard Code for Information Interchange
3. Le Consortium Unicode.
4. Source des illustrations : https://unicode-table.com/

Copyright © 2017 Dunod.

— pour les **codes** 0 **à** 127 (cas les plus fréquents), l'UTF-8 utilise l'octet de la table ASCII ;
— pour les caractères spéciaux (**codes** 128 **à** 2047), quasiment tous nos signes diacritiques, l'UTF-8 utilise 2 octets ;
— pour les caractères spéciaux encore moins courants (**codes** 2048 **à** 65535), l'UTF-8 utilise 3 octets ;
— enfin pour les autres (cas rares), l'UTF-8 en utilise 4.

FIGURE C.1 – Zone ASCII de la table Unicode

FIGURE C.2 – Extrait de la table Unicode

Exemple de l'encodage UTF-8 du caractère Unicode « é » :

Symbole	Code décimal	Code hexadécimal	Encodage UTF-8
é	233	e9	C3 A9

Voici quatre exemples de caractères spéciaux codés en notation hexadécimale ou par nom de caractère Unicode et séparés par le caractère d'échappement tabulation (\t) :

```
>>> print("\u00e9 \t \u03C0 \t \N{Greek Small Letter Pi} \t \u0152")
é       π       π       Œ
```

Applications aux scripts Python

En Python 3, les chaînes de caractères (le type `str()`) sont des chaînes Unicode. Par ailleurs, puisque les scripts Python que l'on produit avec un éditeur sont eux-mêmes des textes, ils sont susceptibles d'être encodés suivant différents formats. Afin que Python sache comment interpréter le

Copyright © 2017 Dunod.

contenu du fichier, il est important d'indiquer l'encodage de caractères utilisé. Si on le précise, ce qui est recommandé, on le note obligatoirement en 1re ou 2e ligne des sources.

Les encodages les plus fréquents sont [1] :

```
# -*- coding: utf8 -*-
```

ou :

```
# -*- coding: latin1 -*-
```

Si on omet de spécifier l'encodage, ou si on en indique un mauvais, on se retrouve avec des textes illisibles, comme ce que l'on peut trouver dans certains courriers électroniques lorsque le logiciel a mal indiqué l'encodage. Un exemple en Python d'encodage mal décodé :

```
>>> print('Caractère préféré : le π'.encode('utf8').decode('latin1'))
CaractÃ¨re prÃ©fÃ©rÃ© : le Ï
```

1. Notons que `utf8` et `latin1` sont des alias de `utf-8` et `latin-1`.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Annexe D

Les expressions régulières

Les expressions régulières [a] fournissent une notation générale très puissante permettant de décrire abstraitement des éléments textuels. Il s'agit d'un vaste domaine pour lequel nous ne proposons qu'une introduction.

a. Ici, l'adjectif *régulier* est employé au sens de *qui obéit à des règles.*

Introduction

Dès les débuts de l'informatique, les concepteurs des systèmes d'exploitation eurent l'idée d'utiliser des *métacaractères* permettant de représenter des modèles généraux. Par exemple, dans un shell Linux ou dans une fenêtre de commande Windows, le symbole `*` [1] remplace une série de lettres, ainsi `*.png` indique tout nom de fichier finissant par l'extension `png`. Les modules standard `glob` et `fnmatch` utilisent la notation des métacaractères.

Depuis ces temps historiques, les informaticiens [2] ont voulu généraliser et standardiser ces notations. On distingue classiquement trois stades dans l'évolution des expressions régulières :

— les expressions régulières **de base** (BRE, *Basic Regular Expressions*) ;
— les expressions régulières **étendues** (ERE, *Extended Regular Expressions*) ;
— les expressions régulières **avancées** (ARE, *Advanced Regular Expressions*).

Trois stades auxquels il convient d'ajouter le support d'Unicode.

Python supporte directement toutes ces évolutions dans son module standard `re` [3].

Les expressions régulières

Une expression régulière [4] se lit (et se construit) de gauche à droite. Elle constitue ce qu'on appelle traditionnellement un *motif* de recherche [5].

Les expressions régulières de base

Elles utilisent six symboles qui, dans le contexte des expressions régulières, acquièrent les significations suivantes :

1. Appelé aussi *joker* (ou *wildcard* en anglais).
2. En particulier le mathématicien Stephen Kleene (1909–1994).
3. Python réutilise l'excellente bibliothèque de traitement des expressions régulières du langage Perl.
4. Souvent abrégée en *regex*.
5. En anglais *search pattern*.

Copyright © 2017 Dunod.

Le point `.` représente une seule instance de n'importe quel caractère sauf le caractère de fin de ligne.

Ainsi l'expression `t.c` représente toutes les combinaisons de trois lettres commençant par `t` et finissant par `c`, comme *tic*, *tac*, *tqc* ou *t9c*, alors que `b.l..` pourrait représenter par exemple *bulle*, *balai* ou *bêler*.

La paire de crochets `[ ]` représente une occurrence quelconque des caractères qu'elle contient.

Par exemple `[aeiouy]` représente une voyelle, et `Duran[dt]` désigne *Durand* ou *Durant*.

Entre les crochets, on peut noter un intervalle en utilisant le tiret [1]. Ainsi, `[0-9]` représente les chiffres de 0 à 9, et `[a-zA-Z]` représente une lettre minuscule ou majuscule.

On peut de plus utiliser l'accent circonflexe en première position dans les crochets pour indiquer « le contraire de ». Par exemple `[^a-z]` représente autre chose qu'une lettre minuscule, et `[^'"]` n'est ni une apostrophe ni un guillemet.

L'astérisque `*` est un *quantificateur*, il signifie aucune ou une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression `ab*` signifie la lettre `a` suivie de zéro ou plusieurs lettres `b`, par exemple *ab*, *a* ou *abbb* et `[A-Z]*` correspond à zéro ou plusieurs lettres majuscules.

L'accent circonflexe `^` est une *ancre*. Il indique que l'expression qui le suit se trouve en début de ligne.

L'expression `^Depuis` indique que l'on recherche les lignes commençant par le mot *Depuis*.

Le symbole dollar `$` est aussi une *ancre*. Il indique que l'expression qui le précède se trouve en fin de ligne.

L'expression `suivante!$` indique que l'on recherche les lignes se terminant par *suivante !*.

L'expression `^Les expressions régulières$` extrait les lignes ne contenant que la chaîne *Les expressions régulières*, alors que `^$` extrait les lignes vides.

La contre-oblique (ou **antislash**) `\` permet d'*échapper* [2] à la signification des métacaractères.

Ainsi `\.` désigne un véritable point, `\*` un astérisque, `\^` un accent circonflexe, `\$` un dollar et `\\` une contre-oblique.

Les expressions régulières étendues

Elles ajoutent cinq symboles qui ont les significations suivantes :

La paire de parenthèses `( )` est utilisée à la fois pour former des sous-motifs et pour délimiter des sous-expressions, ce qui permettra d'extraire des parties d'une chaîne de caractères.

L'expression `(to)*` désignera *to*, *tototo*, etc.

Le signe plus `+` est un *quantificateur* comme `*`, mais il signifie une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression `ab+` signifie la lettre `a` suivie d'une ou plusieurs lettres `b`.

1. Les caractères considérés dans cet intervalle sont ceux dont le code est compris entre les codes des deux caractères délimitatifs.

2. Les informaticiens utilisent fréquemment l'expression « échapper un caractère » pour « le préfixer par le caractère contre-oblique » Par exemple, échapper `n` pour `\n`.

Copyright © 2017 Dunod.

Le point d'interrogation `?`, troisième *quantificateur*, signifie zéro ou une instance de l'expression qui le précède.

Par exemple `écrans?` désigne *écran* ou *écrans*.

La paire d'accolades `{ }` précise le nombre d'occurrences permises pour le motif qui le précède.

Par exemple `[0-9]{2,5}` attend entre deux et cinq chiffres décimaux.

Les variantes suivantes sont disponibles : `[0-9]{2,}` signifie au minimum deux occurrences de chiffres décimaux et `[0-9]{2}` deux occurrences exactement.

La barre verticale `|` représente des choix multiples dans un sous-motif.

L'expression `Duran[dt]` peut aussi s'écrire `(Durand|Durant)`. On pourrait utiliser l'expression `(lu|ma|me|je|ve|sa|di)` dans l'écriture d'une date.

Dans de nombreux outils et langages (dont Python), la syntaxe étendue comprend aussi une série de séquences d'échappement permettant d'identifier des classes entières de caractères [1] :

Séquences d'échappement

`\`	symbole d'échappement
`\e`	séquence de contrôle *escape*
`\f`	saut de page
`\n`	fin de ligne
`\r`	retour chariot
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\d`	classe des chiffres
`\s`	classe des caractères d'espacement
`\w`	classe des caractères alphanumériques ou `_`
`\b`	localisation de début ou de fin de mot
`\D`	négation de la classe `\d`
`\S`	négation de la classe `\s`
`\W`	négation de la classe `\w`
`\B`	négation de la classe `\b`

Note : Par « caractères d'espacement » on entend espace, tabulation et retour à la ligne.

Les expressions régulières avec Python

Le module `re` permet d'utiliser les expressions régulières dans les scripts Python. Les scripts devront donc comporter la ligne :

```
import re
```

Pythonismes

Le module `re` fournit des outils utilisant la programmation objet. Les motifs et les correspondances de recherche seront des objets de la classe `SRE_Pattern` auxquels on pourra appliquer des méthodes.

1. Le standard Unicode définit ces classifications de caractères.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Utilisation des *raw strings*

La syntaxe des motifs comprend souvent le caractère contre-oblique, qui doit être lui-même échappé dans une chaîne de caractères, ce qui alourdit la notation. On peut éviter cet inconvénient en utilisant des « chaînes brutes » préfixées par `r`. Par exemple au lieu de :

```
"\\d\\d? \\w+ \\d{4}"
```

on écrira :

```
r"\d\d? \w+ \d{4}"
```

Les options de compilation

Grâce à un jeu d'*options de compilation* des expressions, il est possible de piloter le comportement des expressions régulières. On utilise pour cela soit des paramètres supplémentaires au constructeur, soit plus fréquemment la syntaxe `(?<drapeau>)` avec les drapeaux suivants :

- **a** correspondance alphanumérique restreinte à l'ASCII (Unicode par défaut) ;
- **i** correspondance alphabétique non sensible à la casse ;
- **L** les correspondances utilisent la *locale*, c'est-à-dire les particularités du pays ;
- **m** appliqué aux chaînes de plusieurs lignes ;
- **s** modifie le comportement du métacaractère point `.`, qui représentera alors aussi le saut de ligne ;
- **u** correspondance alphanumérique Unicode (par défaut) ;
- **x** mode verbeux (permet d'introduire des commentaires).

Voici un exemple de recherche non sensible à la casse :

```
>>> import re
>>> case = re.compile(r"[a-z]+")              # là on est sensible à la casse
>>> print(case.search("Bastille").group())
astille
>>> ignore_case = re.compile(r"(?i)[a-z]+")   # là non : utilisation du drapeau (?i)
>>> print(ignore_case.search("Bastille").group())
Bastille
```

Les motifs nominatifs

Python possède une syntaxe qui permet de nommer des parties de motif délimitées par des parenthèses, ce qu'on appelle un « motif nominatif » :

- syntaxe de création d'un motif nominatif : `(?P<nom_du_motif>)` ;
- syntaxe permettant de s'y référer : `(?P=nom_du_motif)` ;
- syntaxe à utiliser dans un motif de remplacement ou de substitution : `\g<nom_du_motif>`.

Exemples

On propose plusieurs exemples d'extraction de dates historiques telles que `"14 juillet 1789"`.

Copyright © 2017 Dunod.

Extraction simple

Cette chaîne peut être décrite par le motif `\d\d? \w+ \d{4}` que l'on peut expliciter ainsi : « un ou deux entiers décimaux suivis d'un blanc suivi d'un texte alphanumérique d'au moins un caractère suivi d'un blanc suivi de quatre entiers décimaux ».

Détaillons le script :

```
import re

motif_date = re.compile(r"\d\d? \w+ \d{4}")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.group())
```

Après avoir importé le module `re`, on compile l'expression régulière correspondant à une date historique en un objet stocké dans la variable `motif_date`. Puis on applique à ce motif la méthode `search()`, qui retourne un objet de la classe `SRE_Match` donnant l'accès à la première position du motif dans la chaîne et on l'affecte à la variable `corresp`. Enfin on affiche la correspondance complète (en ne donnant pas d'argument à `group()`.

Son exécution produit :

```
14 juillet 1789
```

Extraction des sous-groupes

On aurait pu affiner l'affichage du résultat en modifiant l'expression régulière de recherche de façon à pouvoir capturer les éléments du motif :

```
import re

motif_date = re.compile(r"(\d\d?) (\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print("corresp.group() :", corresp.group())
print("corresp.group(1) :", corresp.group(1))
print("corresp.group(2) :", corresp.group(2))
print("corresp.group(3) :", corresp.group(3))
print("corresp.group(1,3) :", corresp.group(1,3))
print("corresp.groups() :", corresp.groups())
```

Ce qui produit à l'exécution :

```
corresp.group() : 14 juillet 1789
corresp.group(1) : 14
corresp.group(2) : juillet
corresp.group(3) : 1789
corresp.group(1,3) : ('14', '1789')
corresp.groups() : ('14', 'juillet', '1789')
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Extraction des sous-groupes nommés

Une autre possibilité est l'emploi de la méthode `groupdict()`, qui renvoie une liste comportant le nom et la valeur des sous-groupes trouvés (ce qui nécessite de nommer les sous-groupes).

```
import re

motif_date = re.compile(r"(?P<jour>\d\d?) (?P<mois>\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.groupdict())
print(corresp.group('jour'))
print(corresp.group('mois'))
```

Ce qui donne :

```
{'jour': '14', 'mois': 'juillet'}
14
juillet
```

Extraction d'une liste de sous-groupes

La méthode `findall` retourne une liste des occurrences trouvées. Si par exemple on désire extraire tous les nombres d'une chaîne, on peut écrire :

```
>>> import re
>>> nbr = re.compile(r"\d+")
>>> print(nbr.findall("Bastille le 14 juillet 1789"))
['14', '1789']
```

Scinder une chaîne

La méthode `split()` des expressions régulières permet de scinder une chaîne à chaque occurrence du motif de l'expression. Si on ajoute un paramètre numérique `n` (non nul), la chaîne est scindée en au plus `n` éléments :

```
>>> import re
>>> nbr = re.compile(r"\d+")
>>> print("Une coupure à chaque occurrence :", nbr.split("Bastille le 14 juillet 1789"))
Une coupure à chaque occurrence : ['Bastille le ', ' juillet ', '']
>>> print("Une seule coupure :", nbr.split("Bastille le 14 juillet 1789", 1))
Une seule coupure : ['Bastille le ', ' juillet 1789']
```

Substitution au sein d'une chaîne

La méthode `sub()` effectue des substitutions dans une chaîne. Le remplacement peut être une chaîne ou une fonction. Comme on le sait, en Python, les chaînes de caractères sont non modifiables et donc les substitutions produisent de nouvelles chaînes.

Exemples de remplacement d'une chaîne par une autre et des valeurs décimales en leur équivalent hexadécimal :

Copyright © 2017 Dunod.

```
import re

def int2hexa(match):
    return hex(int(match.group()))

anniv = re.compile(r"1789")
print("Premier anniversaire :", anniv.sub("1790", "Bastille le 14 juillet 1789"))

nbr = re.compile(r"\d+")
print("En hexa :", nbr.sub(int2hexa, "Bastille le 14 juillet 1789"))
```

Ce qui affiche :

```
Premier anniversaire : Bastille le 14 juillet 1790
En hexa : Bastille le 0xe juillet 0x6fd
```

Les deux notations suivantes sont disponibles pour les substitutions :

Séquences de substitution

&	contient toute la chaîne recherchée par un motif
\n	contient la sous-expression capturée par la n^{e} paire de parenthèses du motif de recherche ($1 \leq n \leq 9$)

Note : Les méthodes sus-citées `findall()`, `split()`, `sub()` existent aussi sous la forme de fonctions du module `re` mais chaque appel de fonction produit une recompilation de l'expression régulière.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Annexe E

Les messages d'erreur de l'interpréteur

> *« Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité. »*
>
> A. Conan Doyle - *Le signe des quatre*

La chasse aux bogues

Dans les différentes étapes du développement logiciel, de la réflexion sur le problème à traiter à l'exécution du programme résultat, en passant par l'écriture du programme et sa documentation, il est une étape incontournable : la recherche des erreurs, ou *chasse aux bogues* [1].

Certains bogues sont détectés par le langage (erreur de syntaxe, nom ou clé ou index non trouvés...), lors de la phase de compilation d'un module avant son exécution ou bien lors de l'exécution.

D'autres bogues sont des erreurs de logique, qui font que le programme se construit et s'exécute mais ne fait pas ce que l'on veut. Pour ces erreurs d'algorithme, c'est au programmeur d'écrire le code qui détectera les cas invalides et lèvera les exceptions *ad hoc*. [cf. p. 40]

Lecture de code

Lorsqu'on programme, on passe finalement beaucoup plus de temps à lire du code (le nôtre ou celui d'autres développeurs) qu'à en écrire. Avec les autres langages, les professionnels définissent généralement des règles sur l'indentation, le positionnement des caractères de début/fin de bloc, etc. (cf. « coding rules » ou « coding policy »), et il existe souvent des outils qui permettent de faire automatiquement des remises en forme de code [2]. Avec Python, l'utilisation obligatoire de l'indentation force à produire déjà un code lisible. Le bon choix des identificateurs (variables, fonctions, classes...), un découpage cohérent en fonctions de tailles raisonnables chargées de taches précises, la modularité du code, et des commentaires pour expliquer les parties de codes complexes ou les astuces de programmation, aident beaucoup à la compréhension du code et au débogage.

Parfois la simple relecture du code par une tierce personne ou par le programmeur à voix haute [3] permet d'identifier des erreurs ou des incohérences entre ce que l'on veut faire et les instructions que l'on a programmées pour le faire.

1. Le terme anglais *bug*, traduit par « bogue », provient de la description de problèmes dans des systèmes mécaniques, avant l'ère de l'électronique ; il a été repris par les informaticiens avec les premières machines de calcul électromécaniques et a perduré avec les ordinateurs modernes et la programmation.
2. Voir par exemple l'outil `astyle` (http ://astyle.sourceforge.net/).
3. Voir la « méthode du canard en plastique » dans le glossaire [cf. p. 229].

Copyright © 2017 Dunod.

Outils de débogage

Lorsqu'une erreur est détectée par Python ou par votre propre code, une exception est levée qui stoppe l'exécution et remonte par les blocs de traitement d'exception (blocs `except`), qui peuvent traiter/corriger les erreurs, les laisser remonter plus loin ou bien les bloquer. Si aucun bloc de traitement d'exception ne stoppe une erreur, celle-ci *remonte* le code, finit par l'affichage d'une trace d'exécution, le *traceback*, et le programme s'arrête.

Lire un *traceback*

Prenons l'exemple suivant :

```
1. Traceback (most recent call last):
2.   File "/…/moduleprincipal.py", line 3, in <module>
3.     print(module2.fct_mod2_g1(9, 1))
4.   File "/…/module2.py", line 4, in fct_mod2_g1
5.     return 3 * module1.fct_mod1_f2(x, y)
6.   File "/…/module1.py", line 5, in fct_mod1_f2
7.     return 1 + fct_mod1_f1(a, b-1)
8.   File "/…module1.py", line 3, in fct_mod1_f1
9.     return x / y    # Si y vaut 0…
10. ZeroDivisionError: division by zero
```

Trace que l'on va lire **en remontant**.

La dernière ligne (ligne 10) nous indique le type d'erreur qui s'est produit (`ZeroDivisionError`) avec un message d'erreur pour l'utilisateur : `division by zero`.

La ligne au-dessus (ligne 9) nous affiche le contenu de la ligne du programme où l'erreur a été produite, l'expression qui a généré l'erreur.

La ligne précédente (ligne 8) nous indique dans quel fichier Python, à quel numéro de ligne et dans quelle fonction se situe la ligne incriminée.

Les couples de lignes précédents (6+7, 4+5, 2+3) se lisent en remontant et indiquent les lignes du code où se trouvent les appels de fonction qui ont été enchaînés pour arriver à la ligne qui a déclenché l'erreur.

En redescendant, on voit donc la cascade d'appels :

```
print(module2.fct_mod2_g1(9, 0))                # dans moduleprincipal
    > return 3 * module1.fct_mod1_f2(x, y)      # dans module2
        > return 1 + fct_mod1_f1(a, b-1)        # dans module1
            > return x / y    # Si y vaut 0...  # dans module1
```

Le code du module 1 :

```
# module1.py
def fct_mod1_f1(x, y):
    return x / y    # Si y vaut 0...

def fct_mod1_f2(a, b):
    return 1 + fct_mod1_f1(a, b-1)
```

Copyright © 2017 Dunod.

Le code du module 2 :

```
# module2.py
import module1

def fct_mod2_g1(x, y):
    return 3 * module1.fct_mod1_f2(x, y)
```

Et le code du module principal :

```
import module2

print(module2.fct_mod2_g1(3, 4))
print(module2.fct_mod2_g1(9, 1))
```

À partir de là, soit l'erreur est facile à trouver simplement en lisant le code et en traçant à la main les évolutions des valeurs dans les variables, soit il faut « sortir l'artillerie lourde » en utilisant des outils comme indiqués ci-dessous.

Note : Le découpage du code en fonctions autonomes (voire en fonctions *pures* [cf. p. 139]) facilite la mise en place de test systématiques — certaines techniques de développement sont pilotées par l'écriture préalable des tests permettant de valider le code à écrire.

Certains éléments peuvent complexifier la recherche de bogues : erreur se produisant au milieu d'un important jeu de données, erreur liée à un effet de bord (la *mémoire* laissée par un traitement de données précédentes agit sur les données actuelles), erreur liée à un traitement qui est réalisé en parallèle (*multithreading*), erreur liée au temps (au moment de l'exécution). Ces deux derniers cas peuvent produire des erreurs « aléatoires » très difficiles à identifier car difficiles à reproduire.

Note : Si vous utilisez des blocs `try/except` afin de récupérer et traiter les exceptions (levées d'erreurs) dans vos programmes, il est important de :

1. N'intercepter que les erreurs qui vous intéressent en spécifiant leurs classes (sauf besoin, éviter les `except` sans classe ou les `except Exception`).
2. Dans un bloc de traitement d'exception, ne pas bloquer les erreurs que vous ne savez pas complètement corriger, faire un `raise` afin de les retransmettre aux blocs de traitement d'erreur de niveau supérieur.
3. Laisser des traces de ce qui s'est passé dans un fichier texte de log, en incluant les *tracebacks* complets, pour permettre *a posteriori* de voir ce qui s'est passé (débogage *post-mortem*).

Le `print()` à l'ancienne… et les logs

Lorsque le code est assez réduit et ne produit pas trop d'affichages, il est possible d'ajouter des appels à la fonction `print()` afin d'afficher les valeurs des variables intéressantes à certains moments de l'exécution (typiquement un peu avant les lignes qui participent à l'enchaînement conduisant à l'erreur), tracer les passages dans certains blocs conditionnels, tracer les boucles et les données traitées lors des itérations…

Pour les chaînes de caractères, il peut être intéressant d'afficher leur représentation avec la fonction `repr()` qui permet de connaître le contenu exact manipulé.

Le module standard `pprint` et sa fonction `pprint()`, peuvent être utilisés afin d'afficher proprement des conteneurs, éventuellement des conteneurs imbriqués dans d'autres conteneurs.

La lecture, jusqu'à l'erreur, de ces affichages judicieux permet de voir par où le programme est passé et quelles ont été les différentes valeurs manipulées.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

On arrive rapidement à placer dans le code de telles traces, que l'on veut ensuite pouvoir activer/désactiver, envoyer vers un fichier, filtrer... On passe alors par l'utilisation d'instructions conditionnelles pilotées par une ou plusieurs variables globales pour activer/désactiver ces traces, par la comparaison à un « niveau » de trace pour avoir plus ou moins de finesse, ou par l'écriture de fonctions pour avoir un formatage d'informations standard comme la date/heure ou le module d'origine de la trace, l'enregistrement de ces informations dans un fichier.

Et au lieu de réinventer la roue, on finit normalement par adopter un outil de génération et de traitement de logs ; pour Python le module `logging`.

Un petit exemple est donné ci-dessous.

L'exécution de ce code produit un fichier texte de log **`tracecode.log`**[1] :

```
# L'utilisation du module logging
import logging

logging.basicConfig(filename='tracecode.log',
                    level=logging.DEBUG,
                    format='%(asctime)s %(message)s',
                    datefmt='%d/%m/%Y %H:%M:%S')

def fct(a, b, c):
    return(a + b / c)

def calcul(a, b):
    try:
        logging.debug("Appel_f avec %d %d %d", a, b, a)
        return fct(a, b, a)
    except Exception as e:
        logging.exception("Echec à appel_f avec %d %d %d", a, b, a)
        raise

def fonction():
    for a in range(-3, 4):
        for b in range(-3, 4):
            calcul(a, b)

fonction()
```

L'exécution de ce code produit un fichier texte de log : (**`tracecode.log`**[1]) :

```
26/02/2017 20:56:20 Appel_f avec -3 -3 -3
26/02/2017 20:56:20 Appel_f avec -3 -2 -3
...
26/02/2017 20:56:20 Appel_f avec 0 -3 0
26/02/2017 20:56:20 Echec à appel_f avec 0 -3 0
Traceback (most recent call last):
  File "tracecode.py", line 16, in calcul
    return fct(a, b, a)
  File "tracecode.py", line 11, in fct
    return(a + b / c)
ZeroDivisionError: division by zero
```

1. Ce fichier n'est pas écrasé à chaque fois, les nouveaux logs s'enregistrent à la suite des anciens.

Copyright © 2017 Dunod.

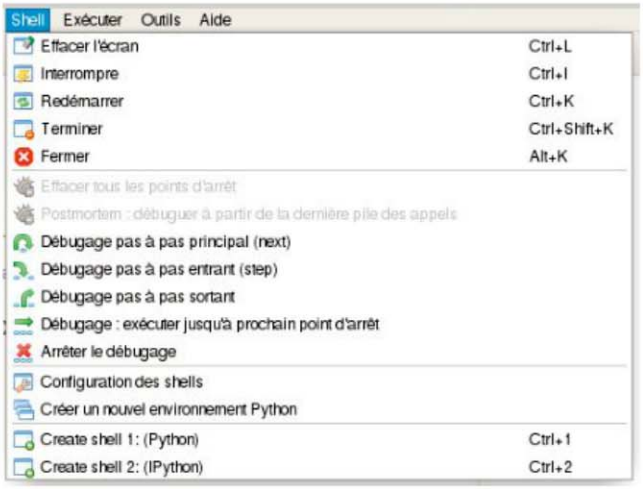
Ce type de fichier peut être assez long.

Cependant, le simple changement de `level=logging.DEBUG` en `level=logging.INFO` ou bien en `level=logging.ERROR` permet de ne plus avoir dans le fichier de log `tracecode.log` que l'indication de l'exception.

Avec le module `logging`, il est possible d'avoir une hiérarchie d'objets *loggers* nommés (pour identifier les objet manipulés ou encore les modules d'origine des traces), différents niveaux de filtrage (`debug`, `information`, `alerte`, `erreur`, `critique`), différents traitements (enregistrement fichier, affichage, envoi vers le système de log de la plateforme, email...). Pour plus de détails, lire la documentation [1].

L'exécution avec un débogueur

Avec `Pyzo`, le débogueur Python se pilote à l'aide de la deuxième partie de commandes du menu Shell :



Vous trouverez ci-dessous un exemple d'utilisation du débogueur de Pyzo. Ce script devrait nous indiquer si un nom commence par une voyelle mais il comporte un petit bogue logique...

```
def comvoy(chaine):
    n = chaine.upper()
    for c in 'aeiouy':
        if n.startswith(c):
            return True
    return False

s = input("Votre nom : ")
if comvoy(s):
    print (s, "commence par une voyelle.")
else:
    print (s, "ne commence pas par une voyelle.")
```

Il faut commencer par placer un *point d'arrêt* dans le programme à un endroit qui nous intéresse. Ici nous le plaçons vers le début du programme principal, juste après la saisie, mais ça peut être au début de la fonction principale ou encore dans une fonction qui pose problème. Pour cela, un simple clic dans la gouttière grise suffit, entre les numéros de ligne et le code source :

1. https://docs.python.org/3/howto/logging.html et https://docs.python.org/3/library/logging.html

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
def comvoy(chaine):
    n = chaine.upper()
    for c in 'aeiouy':
        if n.startswith(c):
            return True
    return False

s = input("Votre nom : ")
if comvoy(s):
    print (s, "commence par une voyelle.")
else:
    print (s, "ne commence pas par une voyelle.")
```

Lorsqu'on lance l'exécution, le script est normalement exécuté jusqu'au premier point d'arrêt, on a donc dans notre exemple effectué la définition de la fonction `comvoy()` puis pu saisir normalement la variable `s` :

```
>>> (executing file "codeadebugger.py")
Votre nom : amandine

(<module>)>>>
```

L'exécution de la ligne où est positionné le point d'arrêt est alors mise en pause, un tiret vert apparaît pour signaler la ligne en attente d'exécution :

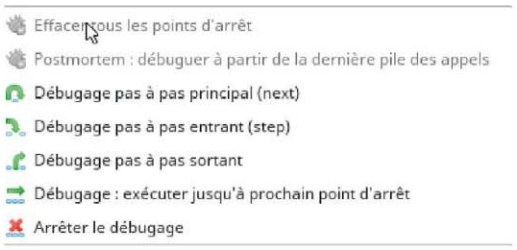
```
s = input("Votre nom : ")
if comvoy(s):
    print (s, "commence par une voyelle.")
```

On peut à ce moment utiliser l'onglet **`Workspace`** de Pyzo pour observer les variables présentes en mémoire et leurs valeurs :

Workspace

Name	Type	Repr
comvoy	function	<function comvoy at 0x7fce2f987048>
s	str	'amandine'

Dans le menu Shell, les commandes d'aide au débogage ont été activées :



Dans l'onglet **`Shells`**, des icônes supplémentaires ont été ajoutées pour ces commandes :

Et dans le shell Python on est aussi passé en mode débogage : il est possible de saisir et évaluer des expressions, de créer de nouvelles variables, de modifier les variables courantes et de faire appel aux commandes du debogueur [1].

1. Saisir la commande *magic* **`db`** pour afficher les commandes du débogueur – les commandes les plus courantes sont directement accessibles *via* les menus et icônes.

Copyright © 2017 Dunod.

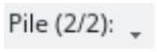
Pas à pas principal : permet d'exécuter la ligne en attente d'exécution et de se remettre en pause à la ligne suivante.

Pas à pas entrant : permet, lorsqu'un appel de fonction se trouve dans l'instruction sur la ligne, d'entrer dans cette fonction en mode pas à pas.

Pas à pas sortant : permet, lorsqu'on est entré dans une fonction en pas à pas entrant, de terminer l'exécution de cette fonction pour en ressortir et se mettre en pause à l'instruction suivant l'appel de cette fonction.

Exécuter jusqu'au prochain point d'arrêt : permet de reprendre l'exécution normalement (sortie du mode pas à pas).

Arrêter le débogage



Affiche le niveau d'appels de fonctions du script en cours de débogage : (le programme principal est le premier niveau) et permet de naviguer entre ces différents niveaux (les variables affichées dans le workspace reflètent les variables locales et globales accessibles dans le niveau d'appel sélectionné).

En cliquant sur le pas à pas entrant, on voit que l'on va exécuter la fonction `comvoy()` :

```
1  def comvoy(chaine):
2      n = chaine.upper()
3      for c in 'aeiouy':
4          if n.startswith(c):
5              return True
6      return False
7
8  s = input("Votre nom : ")
9  if comvoy(s):
10     print (s, "commence par une voyelle.")
11 else:
12     print (s, "ne commence pas par une voyelle.")
```

Et après deux clics sur le pas à pas principal, on est entré dans cette fonction et on obtient :

```
1  def comvoy(chaine):
2      n = chaine.upper()
3      for c in 'aeiouy':
4          if n.startswith(c):
5              return True
6      return False
7
8  s = input("Votre nom : ")
9  if comvoy(s):
10     print (s, "commence par une voyelle.")
11 else:
12     print (s, "ne commence pas par une voyelle.")
```

Workspace

Name	Type	Repr
chaine	str	'amandine'
comvoy	function	<function comvoy at 0x7f34f00d98c8>
n	str	'AMANDINE'
s	str	'amandine'

Pour notre débogage, on peut voir à cette étape que la chaîne `n` sur laquelle on va travailler est entièrement en majuscules, notre recherche basée sur les voyelles en minuscules ne peut qu'échouer, il faut corriger la ligne 2 en utilisant la méthode `lower()`.

Erreurs courantes

Erreur de syntaxe, *SyntaxError*

Cela arrive lorsque Python détecte que la syntaxe du langage n'est pas respectée et ne peut donc analyser le code.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
SyntaxError: invalid syntax
```

On a pu oublier un opérateur (on a fait « des maths » : `y=3x+2` au lieu de `y=3*x+2`), confondre le langage (`$a=5`, `y=x^2`), utiliser l'opérateur d'affectation `=` au lieu de l'opérateur de comparaison `==`...

On peut avoir oublié le caractère `:` qui introduit les blocs contenant des instructions composées `if` / `elif` / `else` / `while` / `for` ou des instructions de gestion des flux d'exceptions `try` / `except` / `finally` ou encore des définitions de classe ou de fonction `class` / `def`...

```
if 1+1 == 2
    print("Vérifié")
```

Il arrive aussi parfois qu'on oublie les guillemets pour terminer une chaîne de caractères, par exemple :

```
msg = "Opération terminée
```

On a alors une erreur de syntaxe :

```
SyntaxError: EOL while scanning string literal
```

Le message indique que Python a rencontré une fin de ligne (`EOL` signifie « End Of Line ») alors qu'il était en train de parcourir le contenu littéral d'une chaîne. Si on désire réellement utiliser des retours à la ligne dans les chaînes, il faut passer aux chaînes triples guillemets [cf. p. 25].

Si l'on oublie la fermeture d'une chaîne triples guillemets mais qu'une autre chaîne triples guillemets est présente ensuite, Python prend l'ouverture de cette chaîne suivante comme la fermeture de la chaîne mal fermée, ce qui provoque généralement une erreur de syntaxe (Python essaie d'analyser le texte de la chaîne...). Généralement la coloration syntaxique dans l'éditeur de code permet de visualiser les chaînes mal fermées et de corriger rapidement ces erreurs.

Si la chaîne triples guillemets mal fermée est la dernière, alors on a l'erreur suivante :

```
SyntaxError: EOF while scanning triple-quoted string literal
```

Le message indique que Python a atteint la fin du fichier (`EOF` signifie « End Of File ») sans rencontrer de fin de chaîne.

À noter que sous Pyzo, le message suivant peut aussi être affiché dans ce cas :

```
Could not run code because it is incomplete
```

`Syntax Error`, mais ma ligne est correcte

Un cas simplifié :

```
def f(x):
    res = [1, 3, 4
    res.append(x)
    return res
```

Là, Python va indiquer une erreur de syntaxe sur la ligne du `res.append()`, pourtant cette ligne est syntaxiquement correcte. L'erreur de syntaxe est à la ligne au-dessus, où il manque un `]` fermant.

Ce genre d'erreur est assez courant et souvent difficile à trouver lorsque l'on débute ; l'oubli peut porter sur tout symbole de fermeture lorsqu'une expression a été ouverte avec `(` ou `[` ou encore `{`. Python permet d'étaler des expressions ainsi ouvertes sur plusieurs lignes jusqu'à ce qu'elles

Copyright © 2017 Dunod.

soient syntaxiquement refermées ; l'erreur ne sera signalée que lorsque l'interpréteur ne réussit plus à analyser.

Lorsqu'une erreur de syntaxe est indiquée pour une ligne et que celle-ci semble correcte, il faut prendre l'habitude de vérifier si les lignes précédentes ferment bien toutes les expressions ouvertes (conteneurs liste, dictionnaire, set, tuple, parenthésage de calculs, appels de fonctions...).

Lors de l'écriture du code, l'utilisation d'un éditeur traçant les symboles qui vont par paires permet de visualiser ces erreurs, par exemple Pyzo souligne le symbole ouvrant/fermant correspondant lorsque le curseur est à côté des `( ) [ ] { }`.

Erreur de type... `TypeError`

Cela arrive lorsqu'on effectue une opération incompatible entre deux types (ou classes) de données.

```
s = "Bonjour"
print(s - "Bon")
```

Python indique dans la dernière ligne du `traceback` l'opération qui a échoué (ici `-`), ainsi que les classes des deux opérandes (ici `str` et `str`).

```
Traceback (most recent call last):
  File "/…/mauvaistype.py", line 2, in <module>
    print (s-"Bon")
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Il vous faut vérifier ces trois informations, opérateur et classes des deux opérandes. Est-ce l'opérateur qui finalement n'existe pas, ou bien (plus souvent) est-ce qu'une des données manipulées n'a pas le type attendu lors de l'exécution ?

Typiquement cela arrive lorsqu'on oublie de faire un transtypage entre des valeurs chaînes et des valeurs numériques avant de faire un calcul, comme dans l'exemple ci-dessous :

```
s = input("age:")
an_nais = 2017 - s
```

À l'exécution :

```
age:55
Traceback (most recent call last):
  File "/…/mauvaistype2.py", line 2, in <module>
    an_nais = 2017 - s
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

L'apparition du `None` (`NoneType`)

Cela peut se retrouver dans un `TypeError` où l'un des opérandes a pris une valeur `None` (avec son type `NoneType`), dans un `AttributeError` où on cherche à accéder à un attribut d'une valeur `None`, etc.

Par exemple avec le programme :

```
def f(a,b,x):
    res = a * x + b
v = 2 * f(2, 5, 3)
```

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

On a lors de l'exécution :

```
Traceback (most recent call last):
  File "<tmp 1>", line 4, in <module>
    v = 2 * f(2, 5, 3)
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

On trouve deux origines principales à ce genre d'erreur :

— l'instruction `return` avec la valeur du résultat à retourner a été oubliée dans une fonction. Par défaut, Python retournant implicitement `None` quand aucune valeur n'est spécifiée (absence de `return` ou `return` sans indication de valeur de retour), ce `None` a été utilisé ;

— une procédure, retournant la valeur `None` car n'étant pas prévue pour retourner une valeur, a été utilisée comme une fonction, ce `None` a été utilisé.

Pour le premier cas, il peut être intéressant, lors de l'apprentissage, de placer systématiquement un `return None` dans les procédures afin d'avoir conscience d'où ils peuvent venir.

Pour le second cas, seule la connaissance des fonctions utilisées et de leur sémantique peut permettre d'identifier l'origine de l'apparition du `None` — cas typique, `ltriee = lst.sort()`, où `sort()` trie la liste *sur place* et retourne `None` (si on veut une copie triée de la liste, on peut utiliser la fonction `ltriee = sorted(lst)`).

Erreur de portée, le global oublié… `UnboundLocalError`

Une erreur assez courante :

```
# Le global oublié.
cptappels = 0
def fct(x):
    print("Valeur:", x)
    cptappels = cptappels + 1
fct(5)
```

À l'exécution, on obtient :

```
Valeur: 5
Traceback (most recent call last):
  File "/…/global_oublie.py", line 6, in <module>
    fct(5)
  File "/…/global_oublie.py", line 5, in fct
    cptappels = cptappels + 1
UnboundLocalError: local variable 'cptappels' referenced before assignment
```

Lors de la compilation du code, Python a identifié en ligne 5 une *affectation* de la variable nommée `cptappels`. En l'absence de directive `global` pour cette variable, elle a été considérée comme un nom obligatoirement local à la fonction. À l'exécution de la ligne 5, Python effectue d'abord le calcul en partie droite, qui nécessite la valeur de `cptappels`. Ce nom est recherché uniquement dans les noms locaux où il n'existe pas.

Solution : si on veut modifier une variable globale par affectation, il faut spécifier une directive `global` pour cette variable dans les fonctions qui la modifient. Si la variable à modifier doit être locale, alors il faut l'initialiser avant de l'utiliser.

Copyright © 2017 Dunod.

```
def fct(x):
    global cptappels
    print("Valeur:", x)
    cptappels = cptappels + 1
```

Changements sur un paramètre par défaut

Vous avez défini une fonction :

```
def calcul(a, b, res=[]):
    for x in range(a,b):
        res.append(x)
    return res
```

Et à l'exécution vous retrouvez des « restes » des appels précédents.

```
print(calcul(1,5))  # [1, 2, 3, 4]
print(calcul(1,7))  # [1, 2, 3, 4, 1, 2, 3, 4, 5, 6]
```

Revoyez tout de suite l'encart Attention concernant les paramètres par défaut [cf. p. 63], ainsi que les arguments mutables et les effets de bord [cf. p. 65], et corrigez votre fonction.

Le nom d'attribut inconnu… `AttributeError`

Ce genre d'erreur arrive lorsqu'on essaie d'utiliser un nom d'attribut (méthode ou variable membre d'un objet) qui n'est pas défini.

```
a = 3
a.arrondi()
```

Python précise dans la dernière ligne du `traceback` la classe de l'objet (données) qui est manipulé (ici `a` contient un entier `int`), ainsi que le nom de l'attribut inconnu (ici `arrondi`).

```
Traceback (most recent call last):
  File "/…/attributinconnu.py", line 2, in <module>
    a.arrondi()
AttributeError: 'int' object has no attribute 'arrondi'
```

À vous de vérifier 1) que vous avez bien une donnée de la classe attendue, et 2) que vous utilisez bien un attribut valide de cette classe.

Si vous avez une erreur d'attribut indiquant qu'un objet de type `NoneType` ne possède pas un attribut particulier, voir annexe E p. 197.

```
AttributeError: 'NoneType' object has no attribute 'xxx'
```

Le nom inconnu… `NameError`

Il peut s'agit d'une variable, fonction, classe… Python cherche un nom (dans les espaces de noms locaux puis globaux puis *builtins*) et ne le trouve pas.

```
print(truc)
```

Il précise dans la dernière ligne du `traceback` le nom qu'il n'a pas trouvé (ici `truc`).

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

```
Traceback (most recent call last):
  File "/…/nominconnu.py", line 1, in <module>
    print(truc)
NameError: name 'truc' is not defined
```

Il peut s'agir d'un nom qui a tout simplement été mal orthographié, ou encore d'un nom qui est défini plus loin lors de l'exécution, donc qui n'existe pas encore lorsque la ligne incriminée est exécutée.

Il peut aussi s'agir d'un nom qui n'est pas accessible dans les espaces de noms courants, par exemple une variable définie localement dans une fonction et qui n'est pas accessible à l'extérieur de cette fonction. Dans ce cas, votre code est à revoir (et vous devez retravailler sur la portée des variables et les espaces de noms).

La clé inconnue… `KeyError`

Son nom est explicite… une clé n'est pas trouvée (dans un dictionnaire, un ensemble…).

```
d = {}
a = d['toto']
```

Python précise dans la dernière ligne du `traceback` la valeur de la clé qu'il n'a pas trouvée dans le conteneur.

```
Traceback (most recent call last):
  File "/…/cleinconnue.py", line 2, in <module>
    a = d['toto']
KeyError: 'toto'
```

À vous de voir ce que contient réellement le conteneur (par exemple en l'affichant juste avant l'opération) et si la clé recherchée est bien celle que vous attendiez.

L'index invalide… `IndexError`

Son nom est aussi explicite : sur un conteneur séquence dont on accède aux éléments par leur position d'index, vous avez utilisé un index hors limites (pour une chaîne, une liste, un tuple…).

```
lst = ['coucou']
a = lst[2]
```

Là, Python ne vous donne malheureusement pas la valeur de l'index utilisé.

```
Traceback (most recent call last):
  File "/home/laurent/docs/python/introprogpython-debugging/indexinconnu.py", line 2,
    in <module> a = lst[2]
IndexError: list index out of range
```

À vous d'afficher la valeur de l'index, éventuellement la taille du conteneur ou son contenu. Lors de vos vérifications, pensez bien à ce que vous avez vu aux paragraphes « Indexation simple » [cf. p. 28] et « Extraction de sous-chaînes » [cf. p. 28], à savoir que les index d'une séquence de N éléments vont de 0 à $N-1$ et que les index négatifs correspondent à des index en partant de la fin.

Copyright © 2017 Dunod.

`IndentationError`

Comme son nom l'indique, le niveau d'indentation d'une ligne n'est pas reconnu par Python, il n'est alors plus capable d'identifier les débuts et fins des blocs d'instructions, qui se basent sur cette indentation syntaxique. Pour éviter ce genre d'erreurs, la première chose à faire est le réglage de l'éditeur de code afin qu'il utilise des espaces à la place des tabulations, et que l'appui sur la touche tabulation insère quatre espaces. Si on utilise un bon éditeur, celui-ci peut afficher des lignes d'indentation (typiquement tous les quatre caractères) et gérer les effacements de « tabulations » en revenant quatre espaces en arrière lorsqu'on efface un espace aligné sur une indentation de bloc.

Cette erreur apparaît généralement sous la forme :

```
IndentationError: unexpected indent
```

Mais on a aussi parfois un message complémentaire lorsque l'erreur est liée à une ligne dont l'indentation a été diminuée par rapport au bloc de la ligne d'instruction qui la précède, mais à un niveau que Python ne peut rattacher à aucun niveau de bloc d'instruction précédent :

```
IndentationError: unindent does not match any outer indentation level
```

Par exemple la dernière ligne du bloc ci-dessous est dans ce cas :

```
if x == 3:
    if y ==2:
        print(y, "vaut deux")
    else:
        print(y)
   print(x)
```

Tableau hiérarchie des exceptions

Nous reprenons ci-dessous le tableau de hiérarchie des exceptions — la notion d'héritage entre les classes d'exceptions est importante pour pouvoir capturer certains types d'erreurs par famille.

Exception	Signification
`BaseException`	✏ La classe de base permettant de structurer la hiérarchie des exceptions
`+– SystemExit`	✏ Un appel à `sys.exit()` a été effectué afin de sortir de l'interpréteur
`+– KeyboardInterrupt`	✏ L'utilisateur a envoyé un signal **`BREAK`** au processus Python (**`Ctrl-C`**)
`+– GeneratorExit`	✏ Utilisé en interne comme mécanisme indiquant qu'une coroutine ou un générateur se termine
`+– Exception`	✏ La base pour les exceptions qui ne sont pas directement liées à une sortie de l'interpréteur, c'est aussi la classe parente pour les exceptions utilisateurs
`   +– StopIteration`	✏ Utilisé en interne comme mécanisme permettant à un itérateur d'indiquer qu'il est arrivé au bout des valeurs à parcourir ; l'instruction de boucle **`for`** intercepte cette exception et termine normalement l'itération

.../...

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

+— Exception	Signification
+— StopAsyncIteration	✏ Même chose pour les itérateurs asynchrones
+— ArithmeticError	✏ Pour toutes les erreurs de calcul
| +— FloatingPointError	✏ Lorsque Python est construit avec certaines options, il peut détecter certaines erreurs de calcul sur les nombres flottants
| +— OverflowError	✏ Pour un dépassement de capacité, cas devenu très improbable pour les entiers car Python passe automatiquement à une représentation sur un nombre variable d'octets (donc une capacité numérique très élevée) lorsque nécessaire. Par ailleurs peu probable pour les nombres flottants car les résultats de ces opérations sont rarement vérifiés
| +— ZeroDivisionError	✏ *No comment* `1/0`
+— AssertionError	✏ Exception levée lorsqu'une instruction `assert` a détecté une condition fausse. Cette instruction est utilisée généralement pour du débogage ou pour outiller du code avec des vérifications que l'on désactive en fonctionnement normal (les instructions `assert` sont ignorées lorsqu'on active l'option `-O` au lancement de Python)
+— AttributeError	✏ Pour un nom d'attribut inconnu [cf. p. 199]
+— BufferError	✏ Lié aux erreurs de gestion ou d'accès à certains types de données plus proches de la machine, qui suivent le « buffer protocol » (types `bytes`, `bytearray`, `array.array`…)
+— EOFError	✏ Lorsque la fin de fichier est atteinte lors d'une lecture sur la console (ou le flux d'entrée standard du programme) par `input()`. Les méthodes de base de lecture des fichiers retournent des chaînes vides plutôt que de lever cette exception
+— ImportError	✏ Quand un import a échoué, le module n'a pas pu être chargé, ou bien un nom importé n'a pas été trouvé (avec `from moduleX import nomY`)
+— ModuleNotFoundError	✏ Un import a échoué car le module demandé n'a pas pu être localisé
+— LookupError	✏ Erreur de recherche dans un conteneur, soit d'index (pour `list`, `str`, `tuple`…), soit de clé (pour `dict`, `set`…)
| +— IndexError	✏ Index numérique hors de séquence [cf. p. 200]
| +— KeyError	✏ Clé non définie [cf. p. 200]
+— MemoryError	✏ Erreur d'allocation mémoire (généralement mémoire pleine)
+— NameError	✏ Un nom local ou global n'a pas été trouvé. Ce nom est précisé dans le message d'erreur
| +— UnboundLocalError	✏ Une variable locale a été utilisée dans une expression avant d'avoir été définie [cf. p. 198]

…/…

Copyright © 2017 Dunod.

+— Exception	Signification
+– `OSError`	✏ Cette exception sert de parente à toutes les erreurs qui sont remontées par le système d'exploitation. On y retrouve des attributs qui permettent d'analyser plus finement l'erreur (`errno`, `winerror`, `strerror`, `filename`, `filename2`...). Toutefois, les classes filles de celle-ci permettent déjà de catégoriser les erreurs en les associant à des opérations spécifiques, sans avoir à se préoccuper des spécificités de la plateforme sur laquelle tourne le programme
| +– `BlockingIOError`	✏ Une opération d'entrée/sortie va conduire à un blocage pour une opération demandée non bloquante
| +– `ChildProcessError`	✏ Une opération sur un processus fils a échoué
| +– `ConnectionError`	✏ Classe de base pour la gestion des connexions (réseau, inter-process...)
| | +– `BrokenPipeError`	✏ Tentative de communication alors que la connexion entre processus par un mécanisme de tubes (*pipes*) ou par un socket réseau a été refermée par le processus pair
| | +– `ConnectionAbortedError`	✏ Tentative de connexion avortée par le processus pair
| | +– `ConnectionRefusedError`	✏ Tentative de connexion refusée par le processus pair
| | +– `ConnectionResetError`	✏ Connexion réinitialisée par le processus pair
| +– `FileExistsError`	✏ Fichier déjà existant
| +– `FileNotFoundError`	✏ Fichier non trouvé (inexistant)
| +– `InterruptedError`	✏ Appel système interrompu par un signal d'interruption (depuis Python 3.5, celui-ci essaie de relancer l'appel système plutôt que de remonter cette exception)
| +– `IsADirectoryError`	✏ Le nom de fichier correspond à un répertoire (l'opération demandée ne peut s'y appliquer)
| +– `NotADirectoryError`	✏ Le nom de fichier ne correspond pas à un répertoire (l'opération demandée ne peut s'y appliquer)
| +– `PermissionError`	✏ Problème de droit d'accès au fichier (ou répertoire). Le problème de droit peut être lié à un répertoire intermédiaire sur le chemin qui doit permettre d'accéder au fichier
| +– `ProcessLookupError`	✏ Processus inexistant
| +– `TimeoutError`	✏ Délai imparti dépassé lors d'un appel système
+– `ReferenceError`	✏ Python permet d'utiliser des « références faibles » (*weak reference*) afin de créer des collections de très nombreux objets dont la mémoire peut être récupérée par le gestionnaire de mémoire « ramasse-miettes ». Cette exception est levée lorsqu'un moyen intermédiaire d'accès (*proxy*) a justement perdu l'objet référencé et ne permet plus d'accéder à son contenu

.../...

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

+— Exception	Signification
+— RuntimeError	✏ Pour les erreurs détectées lors de l'exécution qui ne peuvent pas être plus détaillées, les précisions sont trouvées dans le message d'erreur associé
| +— NotImplementedError	✏ Utilisée généralement dans les classes de base pour les méthodes dont on prévoit qu'elles soient obligatoirement redéfinies par les sous-classes
| +— RecursionError	✏ Une fonction a été appelée récursivement trop de fois et la limite d'appels a été atteinte. Python ne supporte en effet pas la récursion terminale (*tail recursion*), qui permet à certains langages de dérécursiver certaines fonctions ; les appels de fonctions empilés ont donc dû être limités [cf. p. 124]. La récursion peut être d'une cause indirecte (boucle dans les appels de fonctions), ou encore passer par une référence externe (fonction passée en paramètre à une autre fonction)
+— SyntaxError	✏ [cf. p. 195]
| +— IndentationError	✏ [cf. p. 201]
| +— TabError	✏ Généralement un mélange de tabulations et d'espaces dans la définition des blocs d'instructions. Cela a été interdit en Python pour éviter les confusions entre le nombre de blancs considérés par le langage et la représentation qui en est faite par l'éditeur
+— SystemError	✏ Erreur interne de l'interpréteur, qui considère pouvoir tout de même continuer. De telles erreurs devraient être retransmises aux développeurs avec des précisions sur la version de Python (`sys.version`), le message d'erreur et éventuellement un morceau de code qui déclenche l'erreur
+— TypeError	✏ Tentative d'utilisation d'un opérateur ou d'une fonction sur un type de données inapproprié [cf. p. 197]
+— ValueError	✏ Erreur générique lorsqu'une donnée du mauvais type ou d'une valeur inappropriée a été fournie à un opérateur ou à une fonction
| +— UnicodeError	✏ Problème lors de l'encodage/décodage de chaînes de caractères Unicode (les `str` Python 3)
| +— UnicodeDecodeError	✏ Problème lors du décodage octets vers Unicode
| +— UnicodeEncodeError	✏ Problème lors de l'encodage Unicode vers octets
| +— UnicodeTranslateError	✏ Problème lors de l'interprétation d'un caractère
+— Warning	✏ Classe de base d'alertes qui peuvent être remontées par le langage. Elles produisent normalement juste un affichage sur le flux standard d'erreurs, mais Python peut être configuré pour que le mécanisme de traitement des exceptions soit également utilisé pour le traitement des alertes

.../...

Copyright © 2017 Dunod.

+— Exception	Signification
+— `DeprecationWarning`	✏ Alerte qu'une fonctionnalité est en phase d'abandon (pourra avoir disparu et donc générer une erreur dans une version future)
+— `PendingDeprecationWarning`	✏ Alerte qu'une fonctionnalité va passer en phase d'abandon
+— `RuntimeWarning`	✏ Alerte d'un comportement étrange lors de l'exécution
+— `SyntaxWarning`	✏ Alerte d'un comportement étrange à propos de la syntaxe
+— `UserWarning`	✏ Alertes générées par les programmes des utilisateurs
+— `FutureWarning`	✏ Alerte qu'une construction va changer de sens dans une prochaine version
+— `ImportWarning`	✏ Alerte d'une faute probable dans des imports de modules
+— `UnicodeWarning`	✏ Famille d'alertes concernant Unicode
+— `BytesWarning`	✏ Famille d'alertes pour les conteneurs d'octets **`bytes`** et **`bytearray`**
+— `ResourceWarning`	✏ Alerte sur l'utilisation des ressources

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Annexe F

Résumé de la syntaxe

Cette annexe présente des tableaux synthétiques d'emploi des opérateurs et de leur priorité, des chaînes de caractères, des listes, des dictionnaires et des ensembles.

Les opérateurs de Python 3.6 en ordre croissant de précédence

Opérateur	Description
`lambda args : expr`	✏ Créateur de fonction anonyme
`X if Y else Z`	✏ Sélection ternaire (`X` n'est évalué que si `Y` est vrai)
`X or Y`	✏ OU logique : `Y` n'est évalué que si `X` est faux
`X and Y`	✏ ET logique : `Y` n'est évalué que si `X` est vrai
`not X`	✏ Négation logique
`X in S, X not in S`	✏ Opérateurs d'appartenance à un itérable, un ensemble
`X is Y, X is not Y`	✏ Opérateurs d'identité d'objet
`X < Y, X <= Y, X > Y, X >= Y`	✏ Opérateurs de comparaison [1], sous-ensemble et sur-ensemble d'ensembles
`X == Y, X != Y`	✏ Opérateurs d'égalité, de différence
`X | Y`	✏ OU binaire (bit à bit)
`X ^ Y`	✏ OU exclusif binaire (bit à bit)
`X & Y`	✏ ET binaire (bit à bit)
`X << Y, X >> Y`	✏ Décalage binaire de X vers la gauche ou vers la droite de Y bits
`X + Y, X - Y`	✏ Addition/concaténation, soustraction/différence d'ensembles
`X * Y, X @ Y, X / Y, X // Y, X % Y`	✏ Multiplication/répétition, multiplication matricielle, division, division entière, reste de la division entière/formatage de chaînes
`-X, +X`	✏ Négation unaire, identité
`~X`	✏ Complément binaire (inversion des bits)
`X ** Y`	✏ Exponentiation
`X[i]`	✏ Indexation (séquence, dictionnaire, autres)
`X[i:j:k]`	✏ Tranche (les trois indices sont optionnels)
`X(args)`	✏ Appel (fonction, méthode, classe, autres éléments appelables)
`X.attr`	✏ Référence d'attribut
`(....)`	✏ Tuple, expression, expression génératrice
`[....]`	✏ Liste, liste en compréhension
`{....}`	✏ Dictionnaire, ensemble, dictionnaire et ensemble en compréhension

1. Les opérateurs de comparaison peuvent être enchaînés : `x < y < z` est similaire à `x < y and y < z`, sauf que `y` n'est évalué qu'une seule fois dans la première forme.

Copyright © 2017 Dunod.

Remarque : La notation *[xxx]* (avec les *[ ]* en italique !) dénote un xxx optionnel.

Les chaînes de caractères

Syntaxe (chaînes)	Usage
`""`	✏ Construction d'une chaîne vide
`''`	✏ Construction d'une chaîne vide
`str(val)`	✏ Construction d'une chaîne avec la représentation textuelle de la valeur fournie. C'est une conversion en texte de la valeur
`str(val, encoding, errors)`	✏ Construction d'une chaîne à partir du décodage d'une séquence d'octets (`bytes`, `bytearray`, `memoryview`...). On indique dans `encoding` le nom de la méthode d'encodage utilisée (par défaut « utf-8 » — voir le module `codecs`) et dans `errors` la façon de traiter les erreurs de décodage (par défaut `"strict"`, sinon `"ignore"` ou `"replace"`)
`len(s)`	✏ Retourne la longueur (nombre de caractères) de la chaîne `s` (`len` pour *length*).
`s[k]`	✏ Accès au caractère d'index `k` dans la chaîne `s`
`s[déb:fin[:pas]]`	✏ Accès à une sous-chaîne extraite dans la tranche `déb` à `fin` de `s`
`s.count(subs)`	✏ Retourne le nombre d'occurrences (nombre de fois où elle est présente) de la sous-chaîne `subs` (éventuellement un simple caractère) dans la chaîne `s`
`s.count(subs, déb[, fin])`	✏ Idem en se limitant à la tranche entre `déb` et `fin`
`s.index(subs)`	✏ Retourne l'index du premier caractère de la sous-chaîne `subs` (éventuellement un simple caractère) dans la chaîne `s`. Lève une exception `ValueError` si la sous-chaîne n'est pas trouvée
`s.index(subs, déb[, fin])`	✏ Idem en se limitant à la tranche entre `déb` et `fin` (l'index est toujours par rapport au début de la chaîne `s`)
`s.find(subs)`	✏ Retourne l'index du premier caractère de la première occurrence de la sous-chaîne `subs` (éventuellement un simple caractère) dans la chaîne `s`. Retourne `-1` si la sous-chaîne n'est pas trouvée
`s.find(subs, déb[, fin])`	✏ Idem en se limitant à la tranche entre `déb` et `fin` (l'index est toujours par rapport au début de la chaîne `s`)
`s.rfind(subs)`	✏ Retourne l'index du premier caractère de la dernière occurrence (`r` pour *reverse*) de la sous-chaîne `subs` (éventuellement un simple caractère) dans la chaîne `s`. Retourne `-1` si la sous-chaîne n'est pas trouvée
`s.rfind(subs, déb[, fin])`	✏ Idem en se limitant à la tranche entre `déb` et `fin` (l'index est toujours par rapport au début de la chaîne `s`)
`s.capitalize()`	✏ Retourne une version de la chaîne `s` où la première lettre du premier mot est en majuscule, et les autres en minuscules
`s.casefold()`	✏ Retourne une version de la chaîne `s` où les caractères ont été mis en minuscules et adaptés pour une comparaison dans certaines langues (ex. « ß » est converti en « ss » en allemand)
`s.lower()`	✏ Retourne une version de la chaîne `s` où les caractères ont été mis en minuscules

.../...

Copyright © 2017 Dunod.

Syntaxe (chaînes)	Usage
`s.upper()`	✏ Retourne une version de la chaîne `s` où les caractères ont été mis en majuscules
`s.title()`	✏ Retourne une version de la chaîne `s` où chaque mot a sa première lettre en majuscule et les autres en minuscules
`s.swapcase()`	✏ Retourne une version de la chaîne `s` où les caractères ont leur casse (minuscule / majuscule) inversée
`s.rstrip()`	✏ Retourne une version de la chaîne `s` où les caractères blancs (espace, tabulation, retour à la ligne) situés sur la droite (`r` pour *right*) ont été supprimés
`s.rstrip(caracts)`	✏ Idem, en spécifiant les caractères `caracts` à supprimer dans la chaîne
`s.lstrip()`	✏ Retourne une version de la chaîne `s` où les caractères blancs (espace, tabulation, retour à la ligne) situés sur la gauche (`l` pour *left*) ont été supprimés
`s.lstrip(caracts)`	✏ Idem, en spécifiant les caractères `caracts` à supprimer dans la chaîne
`s.strip()`	✏ Retourne une version de la chaîne `s` où les caractères blancs (espace, tabulation, retour à la ligne) situés sur les extrémités (début et fin) ont été supprimés
`s.strip(caracts)`	✏ Idem, en spécifiant les caractères `caracts` à supprimer dans la chaîne
`s.center(larg[, rempl])`	✏ Retourne une version de la chaîne `s` centrée dans une chaîne de `larg` caractères, en remplissant les extrémités par le caractère `rempl` (par défaut espace)
`s.ljust(larg[, rempl])`	✏ Retourne une version de la chaîne `s` alignée à gauche (`l` pour *left*) dans une chaîne de `larg` caractères, en remplissant la fin par le caractère `rempl` (par défaut espace)
`s.rjust(larg[, rempl])`	✏ Retourne une version de la chaîne `s` alignée à droite (`r` pour *right*) dans une chaîne de `larg` caractères, en remplissant le début par le caractère `rempl` (par défaut espace)

Les listes

Les opérations de modification (ou ajout ou suppression) agissent directement **sur** les listes (les listes sont *mutables*). On utilise le terme *item*, qui désigne un élément à une position (qui est un terme anglais, aussi couramment utilisé en informatique).

Syntaxe (listes)	Usage
`[ ]`	✏ Construction d'une liste vide
`[val0, val1, ..., valn]`	✏ Construction d'une liste avec des valeurs (utilisation du séparateur virgule)
`[t(x) for x in séq]`	✏ Construction d'une liste en compréhension, la liste est construite avec une boucle appliquée à une séquence existante `séq`, pour laquelle on applique une transformation `t()` sur chaque élément `x`. Il est possible d'avoir plusieurs niveaux de boucles `for`

.../...

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Syntaxe (listes)	Usage
`[t(x) for x in séq if c(x)]`	✏ Idem, en réalisant en plus un filtrage sur les valeurs de `séq` que l'on veut considérer avec une condition logique `c()` sur chaque élément `x`. Il est possible d'avoir plusieurs `if`
`list()`	✏ Construction d'une liste vide
`list(séquence)`	✏ Construction d'une liste à partir d'une séquence existante. Utilisé entre autre avec le générateur `range()`
`lst1 + lst2`	✏ Construction d'une nouvelle liste par concaténation des items de deux listes `lst1` et `lst2` existantes
`lst.copy()`	✏ Construction d'une nouvelle liste, copie en surface de la liste existante (en surface pour *shallow copy* : les items de la liste qui sont des conteneurs ne sont pas eux-même dupliqués de cette façon). Autre syntaxe : `lst[:]`
`len(lst)`	✏ Retourne la longueur (nombre d'éléments) de la liste `lst` (`len` pour *length).*
`lst[k]`	✏ Accès à la valeur de l'item d'index `k` dans la liste `lst.`
`lst[k] = val`	✏ Modification de l'item à l'index `k` dans la liste `lst`, qui prend la nouvelle valeur `val`
`lst[déb:fin[:pas]]`	✏ Retourne une nouvelle liste de valeurs extraites dans la tranche `déb` à `fin` de `lst`
`lst[déb:fin[:pas]] = séq`	✏ Modification des items situés dans la tranche `déb` à `fin` dans la liste `lst`, qui sont remplacés par les valeurs issues de la séquence `séq`. Les items situés après cette tranche sont tous décalés d'autant de crans que nécessaire, en plus ou en moins
`lst.insert(k, val)`	✏ Insère un item de valeur `val` à l'index `k` de la liste `lst`. Les items qui étaient situés à partir de cet index sont tous décalés d'un cran de plus
`lst.append(val)`	✏ Ajout d'un item de valeur `val` à la fin de la liste `lst`
`lst.extend(séq)`	✏ Ajout d'un ensemble de valeurs issues d'une séquence `séq` à la fin de la liste `lst`
`lst += séq`	✏ Idem
`del lst[k]`	✏ Suppression de l'item à l'index `k` de la liste `lst`. Les items situés après cet index sont tous décalés d'un cran de moins
`del lst[déb:fin[:pas]]`	✏ Suppression des items situés dans la tranche `déb` à `fin` de la liste `lst`. Les items situés après cette tranche sont tous décalés d'autant de crans de moins que nécessaire
`lst.pop()`	✏ Suppression et retour de la valeur du dernier item de la liste `lst`
`lst.pop(k)`	✏ Suppression et retour de la valeur de l'item d'index `k` de la liste `lst`. Les items situés après cet index sont tous décalés d'un cran de moins
`lst.remove(val)`	✏ Recherche du premier item de valeur `val` dans la liste, et suppression de cet item. Les items situés après celui trouvé sont tous décalés d'un cran de moins
`lst.clear()`	✏ Suppression de tous les items de la liste `lst`, qui devient donc vide. Autre syntaxe : `del lst[:]`

.../...

Copyright © 2017 Dunod.

Syntaxe (listes)	Usage
`val in lst`	✏ Teste la présence de la valeur `val` dans la liste `lst` (résultat booléen `True`/`False`)
`val not in lst`	✏ Teste l'absence de la valeur `val` dans la liste `lst` (résultat booléen `True`/`False`)
`lst.count(val)`	✏ Retourne le nombre d'occurrences (nombre de fois où elle est présente) de la valeur `val` dans la liste `lst`
`lst.count(val, déb[, fin])`	✏ Idem en se limitant à la tranche entre `déb` et `fin`
`lst.index(val)`	✏ Retourne l'index de la première occurrence de `val` dans `lst`
`lst.index(val, déb[, fin])`	✏ Idem, en commençant la recherche à partir de l'index de tranche `déb`, en effectuant la recherche jusqu'à l'index de tranche `fin`
`lst.sort()`	✏ Tri des items de la liste `lst` par ordre croissant — les valeurs doivent être comparables. Argument optionnel `reversed=True` pour trier par ordre décroissant
`lst.sort(key=fct)`	✏ Tri des items de la liste `lst` par ordre croissant des valeurs retournées par la fonction `fct()` [1] appliquée à chaque item. Argument optionnel `reversed=True` pour trier par ordre décroissant
`lst.reverse()`	✏ Inversion de l'ordre des items de la liste `lst`
`min(lst)`	✏ Retourne la valeur de l'item le plus petit dans la liste `lst`. Fonction `min()` générique, applicable à toute séquence
`max(lst)`	✏ Retourne la valeur de l'item le plus grand dans la liste `lst`. Fonction `max()` générique, applicable à toute séquence
`sum(lst)`	✏ Retourne la somme numérique des valeurs de la liste `lst`. Ces valeurs doivent être des nombres. Fonction `sum()` générique, applicable à toute séquence

Exceptions courantes rencontrées lors des manipulations sur les listes

Exception	Cause probable
`IndexError`	✏ Une valeur d'index `k` a été utilisée qui est hors des index des éléments de la liste. Par exemple avec une indexation au-delà de la longueur de la liste ou avec `pop()` sur une liste vide
`ValueError`	✏ Une valeur n'a pas été trouvée dans la liste. Par exemple avec `index()` ou avec `remove()`
`TypeError`	✏ Une opération n'a pas pu être effectuée sur des éléments de la liste. Par exemple `sort()` sur une liste qui contient des éléments non comparables, avec une indication supplémentaire : `unorderable types:...` De même pour `min()` ou `max()`, ou encore `sum()` sur une liste qui contient des valeurs non numériques, avec une indication supplémentaire : `unsupported operand type(s) for +:...`

1. Des fonctions d'aide comme `itemgetter()` et `attrgetter()` du module `operator` permettent de trier sur des items ou attributs particuliers.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Les dictionnaires

Syntaxe (dictionnaires)	Usage
`{ }`	✏ Construction d'un dictionnaire vide
`{clé0:val0, clé1:val1, ..., clén:valn}`	✏ Construction d'un dictionnaire avec des clés et valeurs (paires clé-valeur séparées par deux points, séparateur virgule entre les couples)
`{t1(x):t2(x) for x in séq}`	✏ Construction d'un dictionnaire en compréhension. Le dictionnaire est construit avec une boucle appliquée à une séquence existante `séq`, pour laquelle on applique des transformations `t1()` et `t2()` sur chaque élément `x` afin de produire la clé et la valeur. Il est possible d'avoir plusieurs niveaux de boucles `for`
`{t1(x):t2(x) for x in séq if c(x)}`	✏ Idem, en réalisant en plus un filtrage sur les valeurs de `séq` que l'on veut considérer avec une condition logique `c()` sur chaque élément `x`. Il est possible d'avoir plusieurs `if`
`dict()`	✏ Construction d'un dictionnaire vide
`dict(d)`	✏ Construction d'un nouveau dictionnaire copie en surface d'un dictionnaire `d` existant (en surface pour *shallow copy* : les clés et valeurs du dictionnaire existant, qui sont des conteneurs, ne sont pas elles-mêmes dupliquées de cette façon)
`dict(séquence)`	✏ Construction d'un dictionnaire à partir d'une séquence existante de paires. Utilisable avec le générateur `zip()` lorsqu'on dispose de deux séquences séparées pour les clés et les valeurs
`dict(clé0=val0, clé1=val1, ..., clén=valn)`	✏ Construction d'un dictionnaire avec des paires clé-valeur en utilisant une syntaxe d'appel de fonction
`dict.fromkeys(séquence[, défaut])`	✏ Construction d'un dictionnaire à partir des clés dans la séquence, toutes les paires ayant la même valeur `défaut` (par défaut `None`)
`d.copy()`	✏ Construction d'un nouveau dictionnaire copie en surface d'un dictionnaire `d` existant
`len(d)`	✏ Retourne le nombre d'éléments (paires clé-valeur) du dictionnaire `d`
`d[clé]`	✏ Accès à la valeur de la paire pour la clé dans le dictionnaire `d`
`d.get(clé[, défaut])`	✏ Retourne la valeur pour la clé dans le dictionnaire `d`. Si la clé n'est pas présente, retourne `défaut` s'il est fourni ou sinon lève une exception `KeyError`
`d.setdefault(clé[, défaut])`	✏ Retourne la valeur pour la clé dans le dictionnaire `d`. Si la clé n'est pas présente, associe la clé à la valeur `défaut` (par défaut `None`) dans le dictionnaire et retourne cette valeur
`d[clé] = valeur`	✏ Création d'une paire associant clé et valeur dans le dictionnaire `d`. Si une association existe déjà pour cette clé, elle est modifiée avec la nouvelle valeur. Voir aussi `collections.defaultdict`
`d.update(d2)`	✏ Mise à jour du dictionnaire `d` à partir des paires clé-valeur issues du dictionnaire `d2`. Les clés déjà présentes dans `d` voient leurs associations modifiées avec les nouvelles valeurs

.../...

Copyright © 2017 Dunod.

Syntaxe (dictionnaires)	Usage
`d.update(séquence)`	✏ Mise à jour du dictionnaire `d` à partir d'une séquence des paires clé-valeur
`d.update(clé0=val0, clé1=val1, ..., clén=valn)`	✏ Mise à jour du dictionnaire `d` à partir de paires clé-valeur en utilisant une syntaxe d'appel de fonction
`del d[clé]`	✏ Suppression de la paire clé-valeur pour la clé dans le dictionnaire `d`
`d.pop(clé[, défaut])`	✏ Suppression et retour de la valeur pour la clé dans le dictionnaire `d`. Si la clé n'est pas présente, retourne `défaut` s'il est fourni ou sinon lève une exception `KeyError`
`d.popitem()`	✏ Suppression et retour d'une paire au hasard dans le dictionnaire `d`, retournée dans un tuple `(clé, valeur)`. Si le dictionnaire est vide, lève exception `KeyError`
`d.clear()`	✏ Suppression de toutes les paires du dictionnaire `d`
`clé in d`	✏ Teste la présence de `clé` dans le dictionnaire `d` (résultat booléen `True`/`False`)
`clé not in d`	✏ Teste l'absence de `clé` dans le dictionnaire `d` (résultat booléen `True`/`False`)
`d.keys()`	✏ Retourne une vue itérable sur les clés du dictionnaire `d`
`d.values()`	✏ Retourne une vue itérable sur les valeurs du dictionnaire `d`
`d.items()`	✏ Retourne une vue itérable sur les paires (clé, valeur) du dictionnaire `d`
`for k in d:`	✏ Boucle avec la variable `k` sur les clés du dictionnaire `d` (dictionnaire itérable utilisable avec `min()`, `max()`, `sum()`...)
`iter(d)`	✏ Retourne un itérateur sur les clés du dictionnaire `d`

Exception courante rencontrée lors des manipulations sur les dictionnaires

Exception	Cause probable
`KeyError`	✏ Une clé `k` absente du dictionnaire a été utilisée pour chercher une valeur, ou `popitem()` a été utilisé sur un dictionnaire vide

Les ensembles

Syntaxe (ensembles)	Usage
`{val0, val1,... , valn}`	✏ Construction d'un ensemble avec des valeurs (séparateur virgule entre les valeurs)
`set()`	✏ Construction d'un ensemble vide
`set(ens)`	✏ Construction d'un ensemble à partir d'un ensemble `ens` existant (accepte un simple itérable). Les éventuels doublons ne se retrouvent qu'une fois dans le set final
`ens.copy()`	✏ Construction d'un nouvel ensemble copie en surface d'un ensemble `ens` existant (en surface pour *shallow copy* : les valeurs du set existant ne sont pas elles-mêmes dupliquées de cette façon)
`len(ens)`	✏ Retourne le nombre d'éléments de l'ensemble `ens`

.../...

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Syntaxe (ensembles)	Usage
`ens.add(val)`	✏ Ajout d'une valeur `val` dans l'ensemble `ens`
`ens.remove(val)`	✏ Suppression d'une valeur `val` de l'ensemble `ens`. En cas d'absence de l'élément, lève une exception `KeyError`
`ens.discard(val)`	✏ Suppression d'une valeur `val` de l'ensemble `ens` si elle y est présente
`ens.pop()`	✏ Suppression et retour d'une valeur au hasard dans l'ensemble `ens`. Si l'ensemble est vide, lève une exception `KeyError`
`ens.update(ens1, ens2... ensn)`	✏ Mise à jour de l'ensemble `ens` à partir des éléments issus d'un ou plusieurs ensembles (accepte de simples itérables)
`ens |= ens1 |= ens2 ... |= ensn`	✏ Idem, avec des opérateurs entre des ensembles
`ens.intersection_update(ens1, ens2... ensn)`	✏ Mise à jour de l'ensemble `ens` à partir des éléments issus de l'intersection de lui-même et d'un ou plusieurs ensembles (accepte de simples itérables)
`ens &= ens1 &= ens2 ... &= ensn`	✏ Idem, avec des opérateurs entre des ensembles
`ens.difference_update(ens1, ens2... ensn)`	✏ Mise à jour de l'ensemble `ens` en supprimant les valeurs correspondant aux éléments d'un ou plusieurs ensembles (accepte de simples itérables)
`ens -= ens1 -= ens2 ... -= ensn`	✏ Idem, avec des opérateurs entre des ensembles
`ens.symetric_difference_update (ens1)`	✏ Mise à jour de l'ensemble `ens` en ne conservant que les valeurs présentes dans `ens` ou dans `ens1` mais pas dans les deux (accepte un simple itérable)
`ens ^= ens1`	✏ Idem, avec l'opérateur entre des ensembles
`val in ens`	✏ Teste la présence de `val` dans l'ensemble `ens` (résultat booléen `True`/`False`)
`val not in ens`	✏ Teste l'absence de `val` dans l'ensemble `ens` (résultat booléen `True`/`False`)
`ens.isdisjoin(ens1)`	✏ Teste si l'ensemble `ens1` n'a aucun élément en commun avec l'ensemble `ens`
`ens.issubset(ens1)`	✏ Teste si l'ensemble `ens` est un sous-ensemble de l'ensemble `ens1`
`ens <= ens1`	✏ Idem, avec l'opérateur entre des ensembles
`ens < ens1`	✏ Teste si l'ensemble `ens` est un sous-ensemble propre de l'ensemble `ens1` (inclus mais non égal)
`ens.issuperset(ens1)`	✏ Teste si l'ensemble `ens` est un sur-ensemble de l'ensemble `ens1`
`ens >= ens1`	✏ Idem, avec l'opérateur entre des ensembles
`ens > ens1`	✏ Teste si l'ensemble `ens` est un sur-ensemble propre de l'ensemble `ens1` (celui-ci est inclus mais non égal)
`ens.union(ens1, ens2,... ensn)`	✏ Construction d'un nouvel ensemble résultant de l'union de `ens` avec les valeurs issues des éléments d'un ou plusieurs ensembles (accepte de simples itérables)
`ens | ens1 | ens2 ... | ensn`	✏ Idem, avec des opérateurs entre des ensembles

.../...

Copyright © 2017 Dunod.

Syntaxe (ensembles)	Usage
`ens.intersection(ens1, ens2 ,... ensn)`	✏ Construction d'un nouvel ensemble résultant de l'intersection des valeurs de l'ensemble `ens` avec celles issues d'un ou plusieurs ensembles (accepte de simples itérables), l'intersection portant sur les valeurs communes à tous
`ens & ens1 & ens2 ... & enss`	✏ Idem, avec des opérateurs entre des ensembles
`ens.difference(ens1, ens2,... ensn)`	✏ Construction d'un nouvel ensemble à partir de la différence entre les valeurs de l'ensemble `ens` et celles issues d'un ou plusieurs ensembles (accepte de simples itérables). Le nouvel ensemble contient les valeurs de `ens` qui ne sont dans aucun des autres
`ens - ens1 - ens2 ... - enss`	✏ Idem, avec des opérateurs entre des ensembles
`ens.symetric_difference(ens1)`	✏ Construction d'un nouvel ensemble à partir de la différence symétrique entre l'ensemble `ens` et l'ensemble `ens1` (accepte un simple itérable). Le nouvel ensemble contient les valeurs de `ens` et de `ens1` qui ne sont pas dans leur intersection
`ens ^ ens1`	✏ Idem, avec l'opérateur entre des ensembles

Exception courante rencontrée lors des manipulations sur les ensembles

Exception	Cause probable
`KeyError`	✏ Tentative de retrait par `remove()` d'une valeur absente d'un ensemble, ou `pop()` utilisé sur un ensemble vide

Les opérations ensemblistes

Dans le tableau ci-dessous, nous mettons en correspondance les notations Python avec leur équivalent mathématique.

Notons E et F deux ensembles, x un élément quelconque.

Notation Python	Notation mathématique
`len(E)`	$\|E\|$: le cardinal de E
`set()`	$\varnothing$: l'ensemble vide
`x in E`	$x \in E$: l'appartenance
`E < F`	$E \subset F = \{x : x \in E \Rightarrow x \in F\}$ et $E \neq F$: l'inclusion stricte
`E <= F`	$E \subseteq F = \{x : x \in E \Rightarrow x \in F\}$: l'inclusion large
`E & F`	$E \cap F = \{x : x \in E \textit{ et } x \in F\}$: l'intersection
`E | F`	$E \cup F = \{x : x \in E \textit{ ou } x \in F\}$: la réunion
`E - F`	$E \setminus F = \{x : x \in E \textit{ et } x \notin F\}$: la différence
`E ^ F`	$E \Delta F = (E \cup F) \setminus (E \cap F)$: la différence symétrique

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

Copyright © 2017 Dunod.

Bibliographie et webographie

[1] Carella, David, *Règles typographiques et normes. Mise en pratique avec LaTeX*, Vuibert, 2006.

[2] Ziadé, Tarek, *Python : Petit guide à l'usage du développeur agile*, Dunod, 2007.

[3] Ziadé, Tarek, *Programmation Python. Conception et optimisation*, Eyrolles, 2e édition, 2009.

[4] Summerfield, Mark, *Programming in Python* 3, Addison-Wesley, 2e édition, 2009.

[5] Beazley, David M., *Python. Essential Reference*, Addison Wesley, 4e édition, 2009.

[6] Swinnen, Gérard, *Apprendre à programmer avec Python* 3, Eyrolles, 2010.

[7] Kreibich, Jay A., *Using SQLite*, O'Reilly, 2010.

[8] Hellmann, Doug, *The Python Standard Library by Example*, Addison-Wesley, 2011.

[9] Lutz, Mark, *Learnig Python*, O'Reilly, 5e édition, 2013.

[10] Rossant, Cyrille, *IPython Interactive Computing and Data Visualization*, Packt Publishing, 2014.

[11] Driscoll, Michael, *Python 101*, Leanpub, 2014.

[12] Rossant, Cyrille, *Learning IPython for Interactive Computing and Visualization Cookbook*, Packt Publishing, 2e édition, 2015.

[13] Slatkin, Brett, *Learning Effective Python*, Addison Wesley, 2015.

[14] Ramalho, Luciano, *Fluent Python*, O'Reilly, 2015.

[15] Driscoll, Michael, *Python 201*, Leanpub, 2016.

[16] Langtangen, Hans Petter, Linge, Svein, *A Gentle Introduction to Numerical Simulations*, Springer, 2016.

[17] Bitterman, Thomas, *Mastering IPython 4.0*, Packt Publishing, 2016.

[18] Toomey, Dan, *Learning Jupyter*, Packt Publishing, 2016.

[19] Lutz, Mark, *Python précis et concis – Python* 3.4 *et* 2.7, Dunod, 5e édition, 2017.

Copyright © 2017 Dunod.

© Dunod. Toute reproduction non autorisée est un délit.

- Les sites généraux :
  www.python.org
  conda.pydata.org/docs/py2or3.html
  pypi.python.org/pypi
  python.developpez.com/faq
- Interpréteur et EDI spécialisés :
  ipython.org
  jupyter.org
  www.pyzo.org/
- Les outils :
  docs.python.org/3/library/pdb.html
  matplotlib.org
  www.inkscape-fr.org
  code.google.com/p/rst2pdf
  dia-installer.de
  www.tug.org/texworks
- L'encodage :
  sametmax.com/cours-et-tutos
- Une petite référence `NumPy` :
  mathprepa.fr/python-project-euler-mpsi
- Une référence `Tkinter` :
  infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html
- Le lien des liens :
  perso.limsi.fr/pointal

Copyright © 2017 Dunod.

Glossaire et lexique anglais/français

Note : Dans le glossaire, l'acronyme LDP (La Documentation Python) renvoie à une référence dans la documentation officielle Python à l'adresse `http://docs.python.org/3/`.

>>>

Invite Python par défaut dans un shell interactif. Souvent utilisée dans les exemples de code extraits de sessions de l'interpréteur Python.

...

Invite Python par défaut dans un shell interactif, utilisée lorsqu'il faut poursuivre sur plusieurs lignes la saisie d'un bloc indenté, ou à l'intérieur d'une paire de parenthèses, crochets ou accolades.

2to3

Un outil qui essaye de convertir le code Python 2.x en code Python 3.x en gérant la plupart des incompatibilités qu'il peut détecter. `2to3` est disponible dans la distribution `miniconda3`. Voir LDP : Automated Python 2 to 3 code translation.

absolute path (chemin absolu) [cf. p. 54]

Chemin qui commence à partir de la racine du système de fichiers.

abstract base class (ABC) (classe de base abstraite)

Complète le *duck typing* en fournissant un moyen de définir des interfaces. Python fournit de base plusieurs ABC pour les structures de données (module `collections`), les nombres (module `numbers`) et les flux (module `io`). Vous pouvez créer votre propre ABC en utilisant le module `abc`.

accessor (accesseur) [cf. p. 132]

Méthode qui gère l'état d'un attribut, que ce soit en lecture ou en modification.

argument (argument) [cf. p. 61]

Valeur passée à une fonction ou une méthode, affectée à un paramètre local à la fonction. Une fonction ou une méthode peut être appelée à la fois avec des arguments par position et en profitant des valeurs par défaut. Les arguments peuvent être de multiplicité variable : `*` reçoit ou fournit plusieurs arguments par position dans une liste, tandis que `**` joue le même rôle en utilisant les valeurs de paramètres nommés *via* un dictionnaire.
On peut passer toute expression dans la liste d'arguments, et la valeur évaluée est affectée au paramètre local.

assert statement (assertion) [cf. p. 92]

Instruction dont l'expression doit être évaluée à vrai (`True`). En cas d'échec, elle lève une exception `AssertionError`.

attribute (attribut) [cf. p. 92]

Valeur associée à un objet, référencée par un nom et une expression pointée. Par exemple, l'attribut `a` d'un objet `o` peut être référencé `o.a`.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

augmented assignment (affectation augmentée) [cf. p. 23]
Mise à jour d'une variable en utilisant la syntaxe `nom` α`= expression` où α est un opérateur arithmétique équivalent à `nom = nom` α `expression`. Par exemple : `compteur += increment`.

BDFL *Benevolent Dictator For Life* (Dictateur Bienveillant à Vie [cf. p. 7]
Amical surnom de Guido van Rossum, le créateur du langage, dans la communauté Python.

body (corps) [cf. p. 60]
Bloc d'instructions qui définit une fonction ou une méthode.

bytecode (bytecode ou langage intermédiaire) [cf. p. 12]
Le code source Python est compilé en bytecode, représentation interne d'un programme Python dans l'interpréteur. Le bytecode est également rangé dans des fichiers `.pyc` et `.pyo`, ainsi l'exécution d'un même fichier est plus rapide les fois ultérieures (la compilation du source en bytecode peut être évitée). On dit que le bytecode tourne sur une **machine virtuelle** qui, essentiellement, se réduit à une collection d'appels des routines correspondant à chaque code du bytecode.

catch (intercepter) [cf. p. 40]
Le mécanisme des exceptions permet d'intercepter une erreur qu'il fait remonter pour la traiter.

child class (classe fille) [cf. p. 99]
Sous-classe créée en héritant d'une classe mère.

class (classe) [cf. p. 92]
Modèle permettant de créer ses propres objets. Les définitions de classes contiennent des définitions de méthodes qui opèrent sur les instances de classes, ainsi que les définitions d'attributs.

class attribute (attribut de classe) [cf. p. 94]
Attribut lié à une classe. Les attributs de classe sont généralement définis dans une définition de classe, hors des méthodes.

class diagram (diagramme de classe) [cf. p. 94]
Diagramme montrant les relations entre les classes d'un programme. La notation UML est couramment utilisée.

closure (fermeture ou clôture) [cf. p. 128]
Variété de fonction incluse qui utilise des éléments locaux de la fonction enveloppante et qui est renvoyée par celle-ci.

coercion (coercition ou transtypage) [cf. p. 32]
Conversion d'une instance d'un type dans un autre type. Si les types sont compatibles, elle peut être implicite. Si les types sont incompatibles mais que l'opération de transtypage est définie, alors elle peut être réalisée explicitement.

complex number (nombre complexe) [cf. p. 21]
Une extension du système familier des nombres réels dans laquelle tous les nombres sont exprimés comme la somme d'une partie réelle et une partie imaginaire. Les nombres imaginaires sont des multiples réels de l'unité imaginaire (la racine carrée de -1), souvent écrite *i* par les mathématiciens et *j* par les ingénieurs. Python a un traitement incorporé des nombres complexes, qui sont écrits avec cette deuxième notation ; la partie imaginaire est écrite avec un suffixe `j`, par exemple `3+1j`. Pour avoir accès aux équivalents complexes des fonctions du module `math`, utilisez le module `cmath`.

Copyright © 2017 Dunod.

composition (composition) [cf. p. 103]
Type particulier de relation entre deux classes dans lequel la vie des composants est liée à celle de l'agrégat qui les référence.

concatenate (concaténer) [cf. p. 26]
Joindre deux opérandes bout à bout.

context manager (gestionnaire de contexte) [cf. p. 121]
Objet qui contrôle l'environnement protégé indiqué par l'instruction `with` et qui définit les méthodes `__enter__()` et `__exit__()`. Voir la PEP 343.

CPython (Python classique) [cf. p. 8]
Implémentation canonique du langage de programmation Python. Le terme *CPython* est utilisé dans les cas où il est nécessaire de distinguer cette implémentation d'autres comme Jython ou IronPython.

decorator (décorateur) [cf. p. 129]
Fonction appelée pour traiter la définition d'une fonction ou d'une classe, habituellement appliquée comme une transformation utilisant la syntaxe `@wrapper`.
`classmethod`, `staticmethod` et `property` sont des exemples classiques de décorateurs.

deep copy (copie en profondeur ou récursive)
Copie récursive du contenu d'un objet.

data encapsulation (encapsulation de données) [cf. p. 91]
Mécanisme consistant à rassembler les données et les méthodes au sein d'une classe en masquant l'implémentation de l'objet. L'accès aux données se fait par le moyen des méthodes de la classe.

decrement (décrémentation) [cf. p. 38]
Diminution de la valeur d'une variable (généralement par pas de 1).

descriptor (descripteur)
Objet définissant les méthodes `__get__()`, `__set__()` ou `__delete__()`. Lorsqu'un attribut d'une classe est un descripteur, un comportement spécifique est déclenché lors de la consultation de l'attribut. Normalement, l'expression `a.b` consulte l'objet `b` dans le dictionnaire de la classe de `a`, mais si `b` est un descripteur, la méthode `__get__()` (ou `__set__()` pour une affectation) est appelée.
Pour plus d'informations sur les méthodes des descripteurs, voir LDP : Implementing Descriptors.

dictionary (dictionnaire) [cf. p. 49]
Une table associative, dans laquelle des clés arbitraires sont associées à des valeurs. L'accès aux valeurs des objets `dict` ressemble syntaxiquement à celui des objets `list`, mais les clés peuvent être n'importe quels types *hashables*.

docstring (chaîne de documentation) [cf. p. 60]
Chaîne littérale apparaissant comme première expression d'une classe, d'une fonction ou d'un module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et incluse dans l'attribut `__doc__` de la classe, de la fonction ou du module qui la contient. Elle est disponible *via* l'introspection. C'est l'endroit canonique pour documenter un objet.

dot notation (notation pointée) [cf. p. 26]
Syntaxe de résolution de nom dans un espace de noms : `espace.nom`.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

duck typing (typage « comme un canard ») [cf. p. 135]

Style de programmation pythonique dans lequel on détermine le type d'un objet par inspection de ses méthodes et attributs plutôt que par des relations explicites à des types (« s'il ressemble à un canard et fait *coin-coin* comme un canard alors ce doit être un canard »). En mettant l'accent sur des interfaces plutôt que sur des types spécifiques, on améliore la flexibilité du code *via* la substitution polymorphe.

EAFP *Easier to Ask for Forgiveness than Permission* (« plus facile de demander pardon que la permission »)

Ce style courant de programmation en Python consiste à supposer l'existence des clés, des attributs et des droits nécessaires à l'exécution d'un code et à attraper les exceptions qui se produisent lorsque de telles hypothèses se révèlent fausses. C'est un style propre et rapide, caractérisé par la présence d'instructions `try` et `except` pour capturer les cas d'exception. Cette technique contraste avec le style LBYL, courant dans d'autres langages comme le C.

encapsulation (encapsulation) [cf. p. 91]

Mécanisme qui permet d'embarquer les propriétés (attributs et méthodes) d'un objet dans le paradigme de la programmation orientée objet.

expression (expression) [cf. p. 17]

Construction comprenant des littéraux, des noms, des accès aux attributs, des opérateurs ou des appels à des fonctions qui produit une valeur résultante. À l'inverse d'autres langages, toutes les constructions de Python ne sont pas des expressions.

extension module (module d'extension) [cf. p. 69]

Module écrit en C ou en C++ et compilé en binaire machine, utilisant l'API C de Python, qui interagit avec le cœur du langage et avec le code de l'utilisateur. À l'utilisation, Python ne fait pas de distinction entre les modules d'extension et les modules Python.

factory (fabrique) [cf. p. 128]

Une fonction fabrique est une fonction qui crée et renvoie une instance de classe, une fonction, etc.

filter (filtrer) [cf. p. 138]

Traitement qui sélectionne les items d'une séquence satisfaisant certains critères.

first-class function (fonction de première classe) [cf. p. 7]

Se dit des fonctions dans un langage où elles peuvent être instanciées à l'exécution (*runtime*), affectées à des variables, passées en argument ou retournées comme résultats d'autres fonctions.

flag (drapeau)

Variable booléenne donnant la valeur d'une condition.

floor division (division entière) [cf. p. 18]

Division mathématique qui ignore la valeur du reste. L'opérateur de division entière est `//`. Par exemple, l'expression `11//4` est évaluée à `2`, par opposition à la division flottante, qui retourne `2.75`.

flow of execution (flux d'exécution) [cf. p. 35]

Suite de la séquence d'instructions exécutées, en prenant en compte les boucles, les embranchements, les appels de fonctions.

Copyright © 2017 Dunod.

format sequence (séquence de formatage) [cf. p. 30]
Séquence de caractères dans une chaîne de formatage, spécifiant le format à appliquer à une série de valeurs.

function (fonction) [cf. p. 59]
Suite d'instructions qui retourne une valeur à l'appelant. On peut lui passer zéro ou plusieurs arguments qui peuvent être utilisés dans le corps de la fonction. Voir aussi **argument** et **method**.

function call (appel de fonction) [cf. p. 59]
Instruction d'exécution de la fonction.

__future__
Un pseudo-module que les programmeurs peuvent utiliser pour activer les nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur couramment employé. Principalement en Python 2 pour activer certains comportements de Python 3.

garbage collection (ramasse-miettes) [cf. p. 24]
Processus de libération de la mémoire quand elle n'est plus utilisée. CPython exécute cette gestion en comptant les références aux objets en mémoire et en détectant et en cassant les références cycliques.

generator (fonction génératrice) [cf. p. 126]
Une fonction qui renvoie un itérateur. Elle ressemble à une fonction normale, excepté que la valeur de la fonction est rendue à l'appelant en utilisant une instruction `yield` au lieu d'une instruction `return`. Les fonctions génératrices contiennent souvent une ou plusieurs boucles qui « cèdent » des éléments à l'appelant. L'exécution de la fonction est mise en pause au niveau du mot clé `yield`, en renvoyant un résultat, et elle est reprise lorsque l'élément suivant est requis par un appel de la méthode `next()` de l'itérateur.

gather (assembler) [cf. p. 64]
Assemblage des valeurs dans un tuple. On parle aussi d'encapsulation dans un tuple (à ne pas confondre avec l'*encapsulation* de la programmation objet).

generator expression (expression génératrice) [cf. p. 127]
Une expression parenthésée qui produit un générateur. Elle contient une expression normale suivie d'une ou plusieurs boucles `for` définissant une variable de contrôle, un intervalle et zéro ou plusieurs tests `if` permettant des choix.

global interpreter lock (GIL) (verrou global de l'interpréteur)
Le verrou est utilisé par les *threads* (tâches) Python pour assurer qu'un seul *thread* tourne dans la **machine virtuelle CPython** à un instant donné. Il simplifie le fonctionnement de la machine virtuelle Python [cf. p. 12] en garantissant que deux *threads* ne peuvent pas accéder en même temps à une même mémoire. Bloquer l'interpréteur tout entier lui permet d'être *multi-thread safe* aux frais du parallélisme du système environnant.

global variable (variable globale) [cf. p. 66]
Variable définie au niveau principal d'un script. Sa portée s'étend à tout le script.

hashable (hachable) [cf. p. 49]
Un objet est dit hachable s'il a une valeur de hachage constante au cours de sa vie. Cette valeur de hachage, fournie par la méthode `__hash__()` de l'objet, est un calcul d'un entier basé la valeur de l'objet.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

L'« hachabilité » rend un objet propre à être utilisé en tant que clé d'un dictionnaire ou membre d'un ensemble (`set`), car ces structures de données utilisent la valeur de hachage de façon interne.

Tous les objets de base Python non modifiables (*immutables*) sont hachables, alors que certains conteneurs modifiables, comme les listes ou les dictionnaires, ne le sont pas. Les objets instances des classes définies par l'utilisateur sont hachables par défaut, leur valeur de hachage étant leur identité.

header (en-tête) [cf. p. 60]

Dans le contexte de la définition d'une classe, d'une fonction ou d'une méthode, partie constituée du mot clé `class` ou `def`, de l'identificateur et de la suite de la ligne jusqu'au caractère « deux-points ».

higher-order function (fonction d'ordre supérieur)

Se dit d'une fonction qui prend une autre fonction en argument et/ou qui retourne une fonction.

IDLE

IDLE est un environnement de développement intégré pour Python développé par Guido van Rossum. C'est un éditeur basique et un environnement d'interprétation ; il est fourni avec la distribution standard de Python. Excellent pour les débutants, il peut aussi servir d'exemple pour tous ceux qui doivent implémenter une application avec interface utilisateur graphique multi-plateforme avec `tkinter`.

index (indice) [cf. p. 43]

Entier donnant la position d'un item dans une séquence ou dans une chaîne de caractères. L'indice du premier item est 0.

inheritance (héritage) [cf. p. 99]

Mécanisme facilitant la réutilisation par lequel une classe *fille* bénéficie des mêmes caractéristiques que sa classe *mère*.

instance attribute (attribut d'instance) [cf. p. 95]

Attribut lié à une instance d'une classe, c'est-à-dire à un objet de cette classe. Chaque instance possède ses attributs propres, contrairement aux attributs de classe, qui sont partagés par toutes les instances.

immutable (non modifiable)

Un objet avec une valeur fixe. Par exemple, les nombres, les chaînes, les tuples. De tels objets ne peuvent pas être altérés ; pour changer de valeur, il faut créer er affecter un nouvel objet. Les objets non modifiables jouent un rôle important aux endroits où une valeur de hash constante est requise, par exemple pour les clés des dictionnaires.

increment (incrémentation) [cf. p. 38]

Augmentation de la valeur d'une variable (généralement par pas de 1).

instance (instance) [cf. p. 92]

Exemplaire particulier d'une classe. Synonyme d'objet.

instanciate (instancier) [cf. p. 92]

Créer un nouvel objet à partir d'une classe.

item (item)

Élément distinct dans une séquence.

Copyright © 2017 Dunod.

interactive (interactif) [cf. p. 15]

Python possède un interpréteur interactif, ce qui signifie que vous pouvez essayer vos idées et voir immédiatement les résultats. Il suffit de lancer `python` sans argument (éventuellement en le sélectionnant dans un certain menu de votre ordinateur). C'est un moyen puissant pour tester les idées nouvelles ou pour inspecter les modules et les paquetages (pensez à `help(x)`).

interactive mode (mode interactif) [cf. p. 15]

Dans ce mode d'utilisation de Python, les instructions sont directement interprétées dans une boucle d'évaluation.

invariant (invariant)

État qui doit rester constant pendant l'exécution d'une séquence d'instructions.

iteration (itération) [cf. p. 38]

Répétition d'un bloc d'instructions.

interface (interface)

Description générale de l'usage qui doit être fait d'une fonction ou d'une méthode.

interpreted (interprété) [cf. p. 12]

Python est un langage interprété, par opposition aux langages compilés, bien que cette distinction puisse être floue à cause de la présence du compilateur de bytecode. Cela signifie que les fichiers source peuvent être directement exécutés sans avoir besoin de créer préalablement un fichier binaire exécuté ensuite. Typiquement, les langages interprétés ont un cycle de développement et de mise au point plus court que les langages compilés, mais leurs programmes s'exécutent plus lentement. Voir aussi **interactive**.

iterable (itérable) [cf. p. 37]

Un objet conteneur capable de renvoyer ses membres un par un. Des exemples d'*iterable* sont les types séquences (comme les `list`, les `str`, et les `tuple`) et quelques types qui ne sont pas des séquences, comme les objets `dict`, les objets `file` et les objets de n'importe quelle classe que vous définissez avec une méthode `__iter__()` ou une méthode `__getitem__()`.

Les *iterables* peuvent être utilisés dans les boucles `for` (`range()`) et dans beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()`, ...). Lorsqu'un objet *iterable* est passé comme argument à la fonction incorporée `iter()`, il renvoie un itérateur. Cet itérateur est un bon moyen pour effectuer un parcours d'un ensemble de valeurs. Lorsqu'on utilise des *iterables*, il n'est généralement pas nécessaire d'appeler la fonction `iter()` ni de manipuler directement les valeurs en question, l'instruction `for` fait cela automatiquement pour vous en créant une variable temporaire sans nom pour gérer l'itérateur pendant la durée de l'itération. Voir aussi **iterator**, **sequence**, **generator** et **generator expression**.

iterator (itérateur)

Un objet représentant un flot de données. Des appels répétés à la méthode `__next__()` de l'itérateur (ou à la fonction de base `next()`) renvoient des éléments successifs du flot. Lorsqu'il n'y a plus de données disponibles dans le flot, une exception `StopIteration` est lancée. À ce moment-là, l'objet itérateur est épuisé et tout appel ultérieur de la méthode `next()` ne fait que lancer encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui-même. Ainsi un itérateur peut être utilisé dans beaucoup d'endroits où les *itérables* sont acceptés.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

keyword (mot clé) [cf. p. 17]
Mot clé ou mot réservé à la définition du langage. Un mot clé ne peut pas être utilisé comme identifiant.

keyword argument (argument avec valeur par défaut) [cf. p. 63]
Argument précédé par `param_name=` dans l'appel d'une fonction. Le nom du paramètre désigne le nom local dans la fonction, auquel la valeur est affectée. `**` est utilisé pour accepter ou passer un dictionnaire d'arguments en utilisant ses clés avec ses valeurs. Voir **argument**.

lambda function (fonction lambda) [cf. p. 137]
Fonction anonyme définie en ligne, ne comprenant qu'une unique expression dont le résultat fournit la valeur de retour lors de l'appel.

LBYL *Look Before You Leap* (« regarder avant d'y aller »)
Ce style de code teste explicitement les préconditions de validité avant d'effectuer un appel ou une recherche. Ce style s'oppose à l'approche EAFP et est caractérisé par la présence de nombreuses instructions `if`.

list (liste) [cf. p. 43]
Séquence Python de base. En dépit de son nom, elle ressemble plus à ce qui s'appelle « tableau » dans d'autres langages qu'à une liste chaînée puisque l'accès à ses éléments est en $O(1)$ avec un stockage dans un tableau dynamique.

list comprehension (liste en compréhension) [cf. p. 125]
Manière compacte d'effectuer un traitement sur un sous-ensemble d'éléments d'une séquence en renvoyant une liste avec les résultats. Par exemple :

```
result = ["0x%02x" % x for x in range(256) if x % 2 == 0]
```

engendre une liste de chaînes contenant les écritures hexadécimales des nombres pairs de l'intervalle de `0` à `255`. La clause `if` est facultative. Si elle est omise, tous les éléments de l'intervalle `range(256)` seront traités.

local variable (variable locale) [cf. p. 66]
Variable définie dans le corps d'une fonction et visible uniquement dans sa portée.

lookup (recherche) [cf. p. 37]
Opération qui retourne la valeur associée à une clé d'un tableau associatif (dictionnaire).

loop (boucle) [cf. p. 37]
Syntaxe permettant de contrôler la répétition d'un bloc d'instructions.

map (mapper) [cf. p. 138]
Traitement qui effectue une opération sur chaque item d'une séquence pour produire une séquence de résultats.

mapping (tableau associatif) [cf. p. 49]
Un objet conteneur (comme `dict`) qui supporte les recherches par des clés arbitraires en utilisant la méthode spéciale `__get-item__()`.

metaclass (métaclasse)
La classe d'une classe. La définition d'une classe crée un nom de classe, un dictionnaire et une liste de classes de base. La métaclasse est responsable de la création de la classe à partir de ces trois éléments. Beaucoup de langages de programmation orientée objet fournissent une

Copyright © 2017 Dunod.

implémentation par défaut. Une originalité de Python est qu'il est possible de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cela mais, lorsque le besoin apparaît, les métaclasses fournissent des solutions puissantes et élégantes. Elles sont utilisées pour enregistrer les accès aux attributs, pour ajouter des *threads* sécurisés, pour détecter la création d'objets, pour implémenter des singletons et pour bien d'autres tâches.
Des informations complémentaires peuvent être trouvées dans LDP : Customizing class creation.

method (méthode) [cf. p. 97]

Fonction définie dans le corps d'une classe. Appelée comme un attribut d'une instance de classe, la méthode prend cette instance en tant que premier argument (habituellement nommé `self`). Utilisable sans instance, avec les décorateurs `staticmethod` et `classmethod`, les méthodes s'appellent alors directement à partir de la classe. Voir **function** et **nested scope**.

module (module) [cf. p. 69]

Fichier script Python pouvant contenir fonctions, classes et données apparentées offrant un service.

mutable (modifiable)

Les objets modifiables peuvent changer leur valeur sans avoir à passer par une réaffectation (en conservant leur identité). Voir aussi **immutable**.

named tuple (tuple nommé) [cf. p. 79]

Tuple dont les items peuvent aussi être accédés par des noms, comme pour les attributs. Utilise la fonction fabrique `collections.nametuple()`.

namespace (espace de noms) [cf. p. 65]

L'endroit où une variable est conservée. Il y a des espaces de noms locaux, globaux et intégrés et également imbriqués dans les objets. Les espaces de noms contribuent à la modularité en prévenant les conflits de noms. Par exemple, les fonctions `__builtin__.open()` et `os.open()` se distinguent par leur espace de noms. Les espaces de noms contribuent aussi à la lisibilité et à la maintenablité en clarifiant quel module implémente une fonction. Par exemple, en écrivant `random.seed()` ou `itertools.izip()`, on rend évident que ces fonctions sont implémentées dans les modules `random` et `itertools` respectivement.

nested list (liste imbriquée) [cf. p. 46]

Liste de listes.

nested scope (portée imbriquée) [cf. p. 66]

La possibilité de faire référence à une variable d'une définition englobante. Par exemple, une fonction définie à l'intérieur d'une autre fonction peut faire référence à une variable de la fonction extérieure. Notez que les portées imbriquées fonctionnent uniquement pour les références aux variables et non pour leurs affectations, qui concernent toujours la portée imbriquée locale. Les variables locales sont lues et écrites dans la portée la plus intérieure ; les variables globales sont lues et écrites dans l'espace de noms global. L'instruction `nonlocal` permet d'écrire dans la portée englobante.

new-style class (style de classe nouveau)

Vieille dénomination Python 2 pour le style de programmation de classe utilisé en Python 3.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

object (objet) [cf. p. 92]

Toute donnée définie à partir d'une classe, comprenant généralement un état (attributs ou valeurs) et un comportement (méthodes). Également la classe de base ultime du *new-style class*.

operator overloading (surcharge d'opérateur) [cf. p. 98]

Redéfinition du comportement d'un opérateur *via* des méthodes spéciales de sorte qu'il prenne en charge un type défini par le programmeur.

parent class (classe mère) [cf. p. 99]

Classe dont hérite une classe fille.

positional argument (argument de position) [cf. p. 63]

Arguments affectés dans l'ordre de leur position aux noms locaux internes des paramètres d'une fonction ou d'une méthode lors de l'appel. La syntaxe `*` accepte plusieurs arguments de position ou fournit une liste de plusieurs arguments à une fonction. Voir **argument**.

postcondition (postcondition)

Assertion qui doit être satisfaite à la fin de l'exécution d'une séquence de code.

precondition (précondition)

Assertion qui doit être satisfaite au début de l'exécution d'une séquence de code.

property (propriété) [cf. p. 132]

Attribut d'instance permettant d'implémenter les principes de l'encapsulation en utilisant le *protocol descriptor*.

Python3000

Surnom de la version 3 de Python (forgé il y a longtemps, quand la version 3 était un projet lointain). Aussi abrégé « Py3k ».

Pythonic (pythonique)

Qualifie une idée ou un fragment de code plus proche des idiomes du langage Python que des concepts fréquemment utilisés dans d'autres langages. Par exemple, un idiome fréquent en Python est de boucler sur les éléments d'un *iterable* en utilisant l'instruction `for`. D'autres langages n'ont pas ce type de construction et donc les utilisateurs non familiers avec Python utilisent parfois un compteur numérique :

```
for i in range(len(voitures)):
    print(voitures[i])
```

au lieu d'utiliser la méthode claire et pythonique :

```
for voiture in voitures:
    print(voiture)
```

reduce (réduire) [cf. p. 138]

Traitement qui accumule les items d'une séquence en un seul résultat.

refactoring (remaniement)

Processus d'amélioration des qualités du code (clarification des interfaces de fonction, renommage des variables, etc.).

Copyright © 2017 Dunod.

recursive function (fonction récursive) [cf. p. 122]

Fonction dont la définition contient un appel direct ou croisé à elle-même (l'appel croisé provient d'une autre fonction appelée au cours de son exécution).

reference (référence) [cf. p. 21]

Association entre une variable et un objet.

reference count (nombre de références) [cf. p. 24]

Nombre de références d'un objet. Quand le nombre de références d'un objet tombe à zéro, l'objet est désalloué par le *garbage collector*. Le comptage de références n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation de *CPython*.

relative path (chemin relatif) [cf. p. 54]

Chemin qui commence à partir du répertoire courant.

return value (valeur de retour) [cf. p. 62]

Résultat renvoyé par une fonction ou une méthode.

rubber duck debugging (méthode du canard en plastique) [cf. p. 189]

Cette pratique consiste à présenter oralement son code source à un collègue, même non spécialiste, voire à un objet inanimé tel un canard en plastique ! Le simple fait d'expliquer à haute voix ses idées peut aider le programmeur à repérer ses propres erreurs de programmation.

scatter (disperser) [cf. p. 64]

Séparation des valeurs d'une séquence par affectation à une série de variables (aussi décapsulation d'une séquence vers les paramètres d'une fonction).

script mode (mode script) [cf. p. 15]

Dans ce mode d'utilisation de l'interpréteur Python, on enregistre les instructions dans un fichier que l'on exécute ultérieurement.

semantic error (erreur sémantique) [cf. p. 198]

Erreur liée au sens, non à la syntaxe. Un script contenant une erreur sémantique s'exécutera, fera ce que vous lui avez dit de faire..., mais pas ce que vous pensiez qu'il ferait !

sequence (séquence) [cf. p. 43]

Un *iterable* qui offre un accès efficace aux éléments en utilisant des index entiers et les méthodes spéciales `__getitem__()` et `__len__()`. Des types séquences incorporés sont **`list`**, **`str`**, **`tuple`** et **`unicode`**.

shallow copy (copie superficielle)

Copie du contenu d'un objet, y compris les références à des objets inclus, mais sans descendre dans leurs attributs.

singleton (singleton)

Séquence ne contenant qu'un seul item. Classe dont il n'existe qu'une seule instance, le singleton.

slice (tranche) [cf. p. 45]

Objet contenant normalement une partie d'une séquence. Une tranche est créée par une notation indexée utilisant des « : » entre les index début et fin (et pas), comme dans `variable_name[1:3:5]`. La notation crochet utilise des objets *slice* de façon interne.

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

special method (méthode spéciale) [cf. p. 98]
Méthode appelée implicitement par Python pour exécuter une certaine opération sur un type, par exemple une addition. Ces méthodes ont des noms commençant et finissant par deux caractères soulignés. Les méthodes spéciales sont documentées dans LDP : Special method names.

statement (instruction) [cf. p. 35]
Une instruction est une partie d'un bloc de code qui est exécutée. Une instruction est soit une expression, soit une instruction simple, soit une ou plusieurs constructions composées utilisant des mots clés comme `if`, `while`, `for`…

syntax error (erreur syntaxique) [cf. p. 195]
Erreur liée aux règles d'écriture de Python. Un script contenant une erreur de syntaxe ne s'exécutera pas et affichera un *traceback* qui décrira l'erreur.

terminal recursion (récursion terminale) [cf. p. 124]
Une fonction à récursivité terminale est une fonction dans laquelle l'appel récursif est la dernière instruction à être évaluée.

traceback (trace d'appel) [cf. p. 40]
Message complet affiché lors de l'arrêt de l'exécution d'un script suite à une exception (erreur) non capturée.

triple-quoted string (chaîne multiligne) [cf. p. 25]
Chaîne délimitée par trois guillemets (") ou trois apostrophes ('). Elle permet d'inclure des guillemets ou des apostrophes non protégés et peut s'étendre sur plusieurs lignes sans utiliser de caractère de continuation (utile pour les chaînes de documentation).

tuple (tuple ou n-uplet) [cf. p. 45]
Séquence ordonnée immuable d'items.

type (type) [cf. p. 17]
Le type d'un objet Python détermine de quelle sorte d'objet il s'agit ; chaque objet possède un type. Le type d'un objet est accessible grâce à son attribut `__class__` et peut être connu *via* la fonction `type(obj)`.

view (vue)
Les objets retournés par `dict.keys()`, `dict.values()` et `dict.items()` sont appelés des *dictionary views*. Ce sont des « séquences paresseuses [1] » qui laisseront voir les modifications du dictionnaire sous-jacent. Pour forcer un *dictionary view* `dv` à être une liste complète, utiliser `list(dv)`. Voir LDP : Dictionary view objects.

virtual machine (VM) (machine virtuelle) [cf. p. 12]
« Ordinateur » entièrement défini par un programme. La machine virtuelle Python exécute le bytecode généré par le compilateur.

Zen of Python [cf. p. 167]
Liste de principes méthodologiques et philosophiques utiles pour la compréhension et l'utilisation du langage Python. Cette liste peut être obtenue en tapant `import this` dans l'interpréteur Python.

1. L'évaluation paresseuse (en anglais *lazzy evaluation*) est une technique de programmation dans laquelle le programme n'exécute pas de code avant que les résultats de ce code ne soient réellement nécessaires. Le terme « paresseux » étant connoté négativement en français, on parle aussi d'évaluation « retardée ».

Copyright © 2017 Dunod.

Index

Symboles

<< décalage binaire à gauche, 207
<<= opérateur augmenté <<, 23
>> décalage binaire à droite, 207
>>= opérateur augmenté >>, 23
>>> invite Python par défaut, 219
| OU bit à bit, 207
|= opérateur augmenté |, 23
() création de tuple, 45
* multiplication, 18
* répétition de séquence, 26
** élévation à la puissance, 18
**= opérateur augmenté **, 23
*= opérateur augmenté *, 23
+ addition, 18
+ concaténation, 26
+= opérateur augmenté +, 23
- moins unaire, 18
- soustraction, 18
-= opérateur augmenté -, 23
. notation pointée, 26, 30
... invite Python dans un shell interactif, 219
/ division flottante, 18
// division entière, 18
//= opérateur augmenté //, 23
/= opérateur augmenté /, 23
: instruction composée, 35
< inférieur à, 19
<= inférieur ou égal à, 19
== égal à, 19
> supérieur à, 19
>= supérieur ou égal à, 19
[] création de liste, 43
[] opérateur d'indexation, 28
commentaire, 16
% formatage des chaînes, 29
% reste de la division entière, 18
%= opérateur augmenté %, 23
& ET bit à bit, 207
^ OU exclusif bit à bit, 207
^= opérateur augmenté ^, 23
\ lignes de continuation, 35
{} création de dictionnaire, 49
~ NON bit à bit, 207
2to3, 219

A

accesseur, 132, 219
  deleter, 132
  getter, 132
  setter, 132
affectation, 22
  augmentée, 23, 220
agrégation, 103
algorithme, **9**, 79
alternative, 36
annotation, 136
appel, 61
  terminal, 124
arborescence, 54
argument, 61, 219
  de position, 228
  passage par affectation, 61
ASCII, 177
assembler, 223
assertion, 219
association, 102
attribut, 93, 219
  d'instance, 95, 224
  de classe, 95
auto-test, 75

B

base, **19**
batteries included, 7, 77
  avec les piles, 77

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

bibliothèque, 69
  mathématique, 81
  standard, 77
  temps et dates, 79
bloc, 35
Boole, George, 19
boucle, 37, 226
  d'événement, 107
  parcourir, 38
  répéter, 38
builtin, 59
bytecode, 7, **12**, 80, 220

C

C, 7
C++, 7
capture de contexte, 128
CD-ROM, 11
cellule markdown, 175
chaîne, 25
  concaténation, 26
  de documentation, 221
  littérale, 54
    brute, 54
  longueur, 26
  multiligne, 230
  répétition, 26
  séquence d'échappement, 26
chemin d'accès, 54
  absolu, 54, 219
  relatif, 54, 229
classe, 92, 220
  attribut de, 95, 220
  de base abstraite, 219
  diagramme de, 220
  fille, 220
  mère, 228
clôture, 220
codage, **177**
  Unicode, 177
coercition, 220
commentaire, 16
compilateur, 11
compilation
  option de, 184
composition, 102, **103**, 221
concaténer, 221
conception
  association, 102
  dérivation, 102
  graphique, 111
condition terminale, 123
console, 32
conteneur, 37, **43**
copie
  en profondeur, 221
  récursive, 221
  superficielle, 229
corps, 59, 220

D

dates
  gestion des, 79
descripteur, 221
Dictateur Bienveillant à Vie, 220
dictionnaire, 49, 221
  clé, 49
  en compréhension, 126
  valeur, 49
disperser, 229
division, 18
  entière, 18, 222
  flottante, 18
drapeau, 38, 222
duck typing, 135
décorateur, **129**, 129, 221
  post-traitements, 129
  prétraitements, 129
décrémentation, 221
dérivation, **103**
dérécursivation, 124
désérialisation, 141
développement
  dirigé par la documentation, 150

E

échappement, 25
en-tête, 224
encapsulation, 132, 222
  de données, 221
encodage
  UTF-, 177

Copyright © 2017 Dunod.

ensemble, **50**
  en compréhension, 126
entrées-sorties, 32
envoi de messages, 92
erreur
  sémantique, 229
  syntaxique, 230
espace de noms, 227
exception, 40
  gérer une, 40
  intercepter, 220
expression, 17, 222
  génératrice, 127, 223
  régulière, 181
exécution paresseuse, 126

F

fabrique, 222
fermeture, 220
fichier, 53
  binaire, 141
  écriture séquentielle, 55
  encodage des caractères, 54
    ascii, 54
    latin1, 54
    utf8, 54
  fermeture, 54
  gestion de, 53
  lecture séquentielle, 55
  nommage de, 53
  ouverture, 54
  textuel, 53
filtrer, 222
flux
  d'exécution, 222
  d'instructions, 35
fonction, 26, **59**, 223
  d'ordre supérieur, 224
  de première classe, 222
  anonyme, 137
  appel de, 223
  appel terminal de, 124
  directive lambda, 137
  docstring, 59
  en-tête de, 59
  fabrique, 128
  fermeture, 128
  filter, 138
  génératrice, 223
  incluse, 128
  lambda, 226
  map, 138
  PFA : application partielle de, 139
  propre, 139
  pure, 139
  reduce, 138
  récursive, 122, 229
    terminale, 124
formatage, 30
  opérateur, 29
  spécificateurs de, 29
functor, 131

G

gestionnaire, 121
  de contexte, 121, 221
gridder, 109
générateur, 126

H

hachable, 49, 223
hash map, 49
Horner
  méthode de, 123
Hunter, John, 80
héritage, 92, **100**, 224

I

identificateur, 16
  casse, 16
  style, 16
IDLE, 224
imbriqué, 46
immutable, 27
implémentation, 8
incrémentation, 224
indexation
  caractère, 28
  élément, 45
  sous-chaîne, 28
  tranche, 28, 45

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

indice, 224
initialiseur, 98
instance, 92, 224
    attribut d', 95
instancier, 224
instruction, 35, 230
    class, 93
    composée, 35
      boucle, 37
      choix, 36
    conditionnelle, 36
interactif, 225
interface, 225
    graphique, 107
interprété, 225
interpréteur, 11
introspection, 119
invariant, 225
IPython, 80, 82
item, **209**, 224
itérable, 37, 225
itérateur, 225
itération, 225

J

joker, 181
json, 142
Jupyter Notebook, 80, 169

K

Kleene, Stephen, 181
Knuth, Donald, 148

L

lambda, 137
langage
    d'assemblage, 11
    de haut niveau, 11
    intermédiaire, 220
    machine, 11
liste, **44**, 226
    compréhension de, 125
    en compréhension, 125, 226
    imbriquée, 227
literate programming, 148

M

machine virtuelle, 230
mapper, 226
markdown, 175
matplotlib, 85
Meyer, Bertrand, xiii
mode
    interactif, 225
    interprété, 15
    script, 15, 229
modifiable, 227
module, **69**, 227
    d'extension, 222
    import, 70
    math, 20
    matplotlib, 85
    numpy, 83
    principal, 73
    re, 181
Monge
    mélange de, 52
mot réservé, 17
motif
    de recherche, 181
    nominatif, 184
Murphy, 124
métacaractère, 181
métaclasse, 226
méthode, 26, 93, **97**, 227
    du canard en plastique, 229
    spéciale, 98, 230
méthodologie
    objet, 13
    procédurale, 13

N

n-uplet, 230
nombre
    complexe, 220
    de références, 229
non modifiable, 224
notation pointée, 221
numpy, 83

Copyright © 2017 Dunod.

O

objet, 92, 228
  capsule, 93
Oliphant, Travis, 80
opérateur, 19
  de comparaison, 19
  logique, 20
opération, 18
  arithmétique, 18
ordinateur, **11**
Ousterhout, K., 107

P

package, 8, 69, 88
packer, 109
paquet, 88
paramètre, 62
  args, 64
  kwargs, 64
  valeur par défaut, 63
parsing, 77
Perez, Fernando, 80
persistance, 141
PFA, *voir* fonction
pickle, 141
placer, 109
polymorphisme, **100**
portée
  globale, 66
  imbriquée, 227
  locale, 66
postcondition, 228
procédure, 62
programmation orientée objet, 91
  attribut, 93
  classe, 92
  encapsuler, 93
  instance, 92
  méthode, 93
  objet, 92
  polymorphisme, 100
programme, **10**
property, *voir* propriété
propriété, **132**, 228
  accesseur, 132
précondition, 228
PSF, 7
Python
  caractéristiques, 7
  historique, 7
  implémentation, 8
  PEP, 7
pythonique, 126, 228
Pyzo, 169

R

RAM, **11**
ramasse-miettes, 223
recherche, 226
relation, 102
  d'agrégation, 102
remaniement, 228
renommage, 132
reST, 146
  reStructuredText, 146
règle LGI, 66
récursion terminale, 230
réduire, 228
référence, 22, 47, 229
répertoire courant, 54
résolution d'un problème, 8

S

saisie, 38
  filtrée, 38
script, 8
Shell Python, 15
singleton, 229
source, 16
Sphinx, 146
style de classe nouveau, 227
style de programmation
  comme un canard, 222
surcharge, **98**, 98
  d'opérateur, 228
sympy, 86
séquence, 39, **43**, 229
  de formatage, 223
  rupture de, 39
sérialisation, 141

Copyright © 2017 Dunod.
© Dunod. Toute reproduction non autorisée est un délit.

T

table
  ASCII, 177
  Unicode, 177
tableau
  associatif, 49, 226
temps
  gestion du, 79
test, **143**
  fonctionnel, 144
  unitaire, 143
Tim, Peters, 167
trace d'appel, 230
trace d'exécution, *voir* traceback
traceback, 190
tranche, 229
  sous-chaîne, 28
transtypage, 220
transtyper, 32
tuple, **45**, 230
  nommé, 79, 227
type, **17**, 18, 230
  binaire, 32
  bool, 19
  complex, 21
  float, 20
  int, 18

U

Unicode, **177**
USB, 11

V

valeur de retour, 229
van Rossum, Guido, 7
  BDFL, 7
  GvR, 7
variable, 22
  globale, 223
  locale, 226
  nommage, 16
verrou global de l'interpréteur, 223
vue, 230

W

widget, **109**
wildcard, 181
wrapper, 129

X

XML, 80

Z

zen, **167**, 230

Copyright © 2017 Dunod.

Mémento Python 3 (partie 1)

Types de base

entier, flottant, booléen, chaîne, octets

```
int 783  0  -192  0b010 0o642 0xF3
          nul     binaire octal hexa
float 9.23 0.0  -1.7e-6
                        ×10^-6
bool True False
str "Un\nDeux"   Chaîne multiligne :
    retour à la ligne échappé  """X\tY\tZ
    'L\'âme'                   1\t2\t3"""
    ' échappé                  tabulation échappée
bytes b"toto\xfe\775"
          hexadécimal  octal
```

☝ *immutables*

Types conteneurs

▪ **séquences ordonnées**, accès par index rapide, valeurs répétables

```
list  [1,5,9]   ["x",11,8.9]   ["mot"]   []
tuple (1,5,9)   11,"y",7.4     ("mot",)  ()
```

Valeurs non modifiables (immutables) ☝ *expression juste avec des virgules →* `tuple`

`str bytes` *(séquences ordonnées de caractères / d'octets)* `""` `b""`

▪ **conteneurs clés**, sans ordre *a priori*, accès par clé rapide, chaque clé unique

```
dictionnaire dict {"clé":"valeur"}   dict(a=3,b=4,k="v")   {}
(couples clé/valeur) {1:"un",3:"trois",2:"deux",3.14:"π"}
ensemble     set {"clé1","clé2"}     {1,9,3,0}            set()
```

☝ *clés=valeurs hachables (types base, immutables…)* `frozenset` *ensemble immutable* *vides*

Identificateurs

pour noms de variables, fonctions, modules, classes…

`a…zA…Z_` suivi de `a…zA…Z_0…9`

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

☺ `a toto x7 y_max BigOne`

☹ ~~`8y and for`~~

Affectation de variables

`=`

☝ **association** d'un *nom* à une *valeur*

1) évaluation de la valeur de l'expression de droite
2) affectation dans l'ordre avec les noms de gauche

```
x=1.2+8+sin(y)
a=b=c=0      affectation à la même valeur
y,z,r=9,-7.6,0   affectations multiples
a,b=b,a      échange de valeurs
a,*b=seq  } dépaquetage de séquence en
*a,b=seq  } élément et liste
x+=3   incrémentation ⇔ x=x+3      et *= /= %= …
x-=2   décrémentation ⇔ x=x-2
x=None valeur constante « non défini »
del x  suppression du nom x
```

Conversions

`type(`*expression*`)`

```
int("15") → 15
int("3f",16) → 63      spécification de la base du nombre entier en 2nd paramètre
int(15.56) → 15        troncature de la partie décimale
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6    arrondi à 1 décimale (0 décimale → nb entier)
bool(x)  False pour x nul, x conteneur vide, x None ou False ; True pour autres x
str(x)→"…"  chaîne de représentation de x pour l'affichage (cf. Formatage, en partie 2)
chr(64)→'@'  ord('@')→64   code ↔ caractère
repr(x)→"…"  chaîne de représentation littérale de x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"trois"),(1,"un")]) → {1:'un',3:'trois'}
set(["un","deux"]) → {'un','deux'}
str de jointure et séquence de str → str assemblée
    ':'.join(['toto','12','pswd']) → 'toto:12:pswd'
str découpée sur les blancs → list de str
    "des mots  espacés".split() → ['des','mots','espacés']
str découpée sur str séparateur → list de str
    "1,4,8,2".split(",") → ['1','4','8','2']
séquence d'un type → list d'un autre type (par liste en compréhension)
    [int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Indexation des conteneurs séquences

pour les listes, tuples, chaînes de caractères, bytes…

index négatif	−5	−4	−3	−2	−1
index positif	0	1	2	3	4
`lst=`	[10,	20,	30,	40,	50]
tranche positive	0	1	2	3	4	5
tranche négative	−5	−4	−3	−2	−1	

Nombre d'éléments `len(lst)→5`

☝ **index à partir de 0** (de 0 à 4 ici)

Accès individuel aux **éléments** par `lst[`*index*`]`

```
lst[0]→10   ⇒ le premier   lst[1]→20
lst[-1]→50  ⇒ le dernier   lst[-2]→40
```

Sur les séquences modifiables (`list`), suppression avec `del lst[3]` *et modification par affectation* `lst[4]=25`

Accès à des **sous-séquences** par `lst[`*tranche début*`:`*tranche fin*`:`*pas*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]     lst[::-2]→[50,30,10]         lst[-3:-1]→[30,40]  lst[3:]→[40,50]
lst[::2]→[10,30,50]      lst[:]→[10,20,30,40,50]  copie superficielle de la séquence
```

Indication de tranche manquante → à partir du début / jusqu'à la fin.
Sur les séquences modifiables (`list`), suppression avec `del lst[3:5]` *et modification par affectation* `lst[1:4]=[15,25]`

Logique booléenne

Comparateurs : `<` `>` `<=` `>=` `==` `!=`
(résultats booléens) ≤ ≥ = ≠

`a and b` et logique — *les deux en même temps*

`a or b` ou logique — *l'un ou l'autre ou les deux*

☝ piège : `and` *et* `or` *retournent la valeur de* `a` *ou de* `b` *(selon l'évaluation au plus court).*
⇒ *s'assurer que* `a` *et* `b` *sont booléens.*

`not a` non logique

`True` `False` constantes Vrai/Faux

Blocs d'instructions

instruction parente :
indentation !
bloc d'instructions 1…
⋮
instruction parente :
bloc d'instructions 2…
⋮
instruction suivante après bloc 1

☝ *régler l'éditeur pour insérer 4 espaces à la place d'une tabulation d'indentation.*

Modules & Imports

module `truc` ⇔ *fichier* `truc.py`

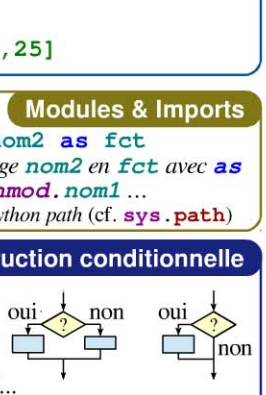
```
from monmod import nom1,nom2 as fct
    → accès direct a nom1, renommage nom2 en fct avec as
import monmod  → accès via monmod.nom1 …
```

☝ modules et packages cherchés dans le *python path* (cf. `sys.path`)

Instruction conditionnelle

un bloc d'instructions exécuté, uniquement ***si*** *sa condition est vraie*

`if` *condition logique* `:`
→ *bloc d'instructions*



Combinable avec des *sinon si, sinon si…* et un seul *sinon* final. Seul le bloc de la première condition trouvée vraie est exécuté.

```
if age<=18:
  etat="Enfant"
elif age>65:
  etat="Retraité"
else:
  etat="Actif"
```

☝ *avec une variable* `x` *:*
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Maths

☝ *nombres flottants… valeurs approchées !*

Opérateurs : `+` `-` `*` `/` `//` `%` `**`

Priorités `(…)` × ÷ entière ↑ reste ÷ ↑ a^b

`@` → × matricielle *(python3.5+* `numpy`*)*

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

☝ *priorités usuelles*

angles en radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0   √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

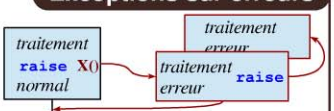
modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, etc.

Exceptions sur erreurs

Signalisation sur détection :
`raise` *ExcClass(…)*

Traitement :

```
try:
→ bloc traitement normal
except ExcClass as e:
→ bloc traitement erreur
```



☝ *bloc* `finally` *pour traitements finaux dans tous les cas.*

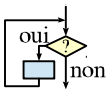
Copyright © 2017 Dunod.

Mémento Python 3 (partie 2)

Instruction boucle conditionnelle

bloc d'instructions exécuté ***tant que*** *la condition est vraie*

```
while condition logique :
    bloc d'instructions
```




```
s = 0          } initialisations avant la boucle
i = 1          } condition avec au moins une valeur
while i <= 100:        variable (ici i)
    s = s + i**2
    i = i + 1          faire varier la variable de condition !
print("somme:",s)
```

Algo : $s = \sum_{i=1}^{i=100} i^2$

attention aux boucles sans fin !

Contrôle de boucle

`break` sortie immédiate
`continue` itération suivante
bloc `else` en sortie *normale* de boucle.

Affichage

```
print("v=",3,"cm :",x,",",y+4)
```

éléments à afficher : valeurs littérales, variables, expressions

Options de `print` :

- `sep=" "` séparateur d'éléments, défaut espace
- `end="\n"` fin d'affichage, défaut fin de ligne
- `file=sys.stdout` print vers fichier, défaut sortie standard

Saisie

```
s = input("Directives:")
```

`input` retourne toujours une **chaîne**, la convertir vers le type désiré (cf. encadré *Conversions*, en partie 1).

Opérations génériques sur conteneurs

`len(c)` → nb d'éléments
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` *copie triée*
Note : pour dictionnaires et ensembles, ces opérations travaillent sur les ***clés****.*
`val in c` → booléen, opérateur `in` de test de présence (`not in` d'absence)
`enumerate(c)` → *itérateur* sur (index, valeur)
`zip(c1,c2...)` → *itérateur* sur tuples contenant les éléments de même index des c_i
`all(c)` → `True` si **tout** élément de `c` évalué vrai, sinon `False`
`any(c)` → `True` si **au moins un** élément de `c` évalué vrai, sinon `False`

Spécifique aux ***conteneurs de séquences ordonnées*** *(listes, tuples, chaînes, bytes...)*

`reversed(c)` → *itérateur inversé* `c*5` → duplication `c+c2` → concaténation
`c.index(val)` → *position* `c.count(val)` → *nb d'occurences*

```
import copy
copy.copy(c) → copie superficielle (1er niveau) du conteneur
copy.deepcopy(c) → copie en profondeur (récursive) du conteneur
```

Opérations sur listes

modification de la liste originale

`lst.append(val)`	ajout d'un élément à la fin
`lst.extend(seq)`	ajout d'une séquence d'éléments à la fin
`lst.insert(idx,val)`	insertion d'un élément à une position
`lst.remove(val)`	suppression du premier élément de valeur *val*
`lst.pop([idx])` → *valeur*	supp. & retourne l'item à l'index (sinon le dernier)
`lst.sort()` `lst.reverse()`	tri / inversion de la liste sur place

Opérations sur dictionnaires

`d[clé]=valeur` `d.clear()`
`d[clé]` → *valeur* `del d[clé]`
`d.update(d2)` } mise à jour/ajout des couples
`d.keys()` `d.values()` `d.items()` } → *vues itérables sur les clés / valeurs / couples*
`d.pop(clé[,défaut])` → *valeur*
`d.popitem()` → *(clé,valeur)*
`d.get(clé[,défaut])` → *valeur*
`d.setdefault(clé[,défaut])` → *valeur*

Opérations sur ensembles

Opérateurs :
`|` → union (caractère barre verticale)
`&` → intersection
`-` `^` → différence/diff. symétrique
`<` `<=` `>` `>=` → relations d'inclusion
Existent aussi sous forme de méthodes.

`s.update(s2)` `s.copy()`
`s.add(clé)` `s.remove(clé)`
`s.discard(clé)` `s.clear()`
`s.pop()`

Fichiers

stockage de données sur disque, et relecture

```
f = open("fic.txt","w",encoding="utf8")
```

- **variable** fichier pour les opérations
- **nom** du fichier sur le disque (+ chemin…) — Cf. modules `os`,`os.path`,`pathlib`
- **mode** d'ouverture
  - `'r'` lecture (*read*)
  - `'w'` écriture (*write*)
  - `'a'` ajout (*append*)
  - …`'+'` `'x'` `'b'` `'t'`
- **encodage** des caractères pour les **fichiers textes** : `utf8` `ascii` `latin1` …

en **écriture**

```
f.write("coucou")
f.writelines(list lignes)
```

en **lecture**

lit chaîne vide si fin de fichier

```
f.read([n])        → caractères suivants
                     si n non spécifié, lit jusqu'à la fin !
f.readlines([n])   → list lignes suiv.
f.readline()       → ligne suivante
```

par défaut mode texte `t` *(lit/écrit* `str`*), mode binaire* `b` *possible (lit/écrit* `bytes`*).* ***Convertir de/vers le type désiré !***

`f.close()` *ne pas oublier de* ***refermer le fichier*** *après son utilisation !*

`f.flush()` écriture du cache `f.truncate([taille])` retaillage

lecture/écriture progressent séquentiellement dans le fichier, modifiable avec :

`f.tell()` → *position* `f.seek(position[,origine])`

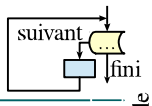
Très courant : ouverture en *bloc gardé* (fermeture automatique) et boucle de lecture des lignes d'un fichier texte :

```
with open(...) as f:
    for ligne in f :
        # traitement de ligne
```

Instruction boucle itérative

bloc d'instructions exécuté ***pour chaque*** *élément d'un conteneur ou d'un itérateur*

```
for var in séquence :
    bloc d'instructions
```



Parcours des **valeurs** d'un conteneur

```
s = "Du texte"   } initialisations avant la boucle
cpt = 0
```

variable de boucle, ***affectation gérée par l'instruction*** `for`

```
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```

Algo : comptage du nombre de `e` *dans la chaîne.*

boucle sur dict/set ⇔ boucle sur séquence des clés ; utilisation des *tranches* pour parcourir un sous-ensemble d'une séquence

Parcours des **index** d'un conteneur séquence

- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11,18,9,12,23,4,17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        perdu.append(val)
        lst[idx] = 15
print("modif:",lst,"-modif:",perdu)
```

Algo : bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

bonne habitude : ne pas modifier la variable de boucle

Parcours simultané **index** et **valeurs** de la séquence :

```
for idx,val in enumerate(lst):
```

Séquences d'entiers

`range([début,] fin [,pas])`

début défaut 0, *fin* non compris, *pas* signé et défaut 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11
`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10
`range(len(séq))` → *séquence des index des valeurs dans séq*

range fournit une séquence immutable d'entiers construits au besoin

Définition de fonction

nom de la fonction (identificateur) — paramètres nommés

```
def fct(x,y,z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```

`return res` ← valeur résultat de l'appel, si pas de résultat calculé à retourner : `return None`

les paramètres et toutes les variables de ce bloc n'existent que dans le bloc et pendant l'appel à la fonction (penser "boîte noire")

Avancé : `def fct(x,y,z,*args,a=3,b=5,**kwargs):`
`*args` ↣ *nb variables d'arguments positionnels (*`tuple`*),* *`a=3`* ↣ *valeurs par défaut,* *`**kwargs`* ↣ *nb variable d'arguments nommés (*`dict`*).*

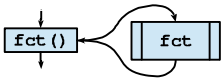
Appel de fonction

```
r = fct(3,i+2,2*i)
```

stockage/utilisation de la valeur de retour — *une valeur d'argument par paramètre*

c'est l'utilisation du nom de la fonction avec les parenthèses qui fait l'appel

*Avancé: *séquence **dict*



Opérations sur chaînes

`s.startswith(prefix[,début[,fin]])` `s.endswith(suffix[,début[,fin]])` `s.strip([caractères])`
`s.count(sub[,début[,fin]])` `s.partition(sep)` → *(avant,sep,après)*
`s.index(sub[,début[,fin]])` `s.find(sub[,début[,fin]])`
`s.is...()` *tests sur les catégories de caractères (ex.* `s.isalpha()`*)*
`s.upper()` `s.lower()` `s.title()` `s.swapcase()`
`s.casefold()` `s.capitalize()` `s.center([larg,rempl])`
`s.ljust([larg,rempl])` `s.rjust([larg,rempl])` `s.zfill([larg])`
`s.encode(codage)` `s.split([sep])` `s.join(séq)`

Formatage

directives de formatage — valeurs à formater

```
"modele{} {} {}".format(x,y,r) → str
"{sélection:formatage!conversion}"
```

- **Sélection** :
  `2`
  `nom`
  `0.nom`
  `4[clé]`
  `0[2]`
- **Formatage** :

Exemples :

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="L'ame")
→'"L\'ame"'
```

car-rempl. *alignement* *signe* *larg.mini*`.`*précision~larg.max* *type*

`<` `>` `^` `=` — `+` `-` *espace* — `0` au début pour remplissage avec des 0

entiers : `b` binaire, `c` caractère, `d` décimal (défaut), `o` octal, `x` ou `X` hexa…
flottant : `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut),
chaîne : `s` … `%` pourcentage

- **Conversion** : `s` (texte lisible) ou `r` (représentation littérale)